#1: Desk Accessories and System Resources

See also: The Resource Manager

Written by: Bryan Stearns
Updated:

February 25, 1985
March 1, 1988

This note formerly described a strategy for dealing with system resources from desk accessories. We no longer recommend calling `ReleaseResource` or `DetachResource` for a system resource. When you are done with a system resource, leave it alone; do not try to dispose or release it.

#2: Compatibility Guidelines

Written by: Cary Clark, Scott Knaster — January 21, 1986
Modified by: Louella Pizzuti — February 9, 1987
Updated: March 1, 1988

Apple has many enhancements planned for the Macintosh family of computers. To help ensure your software's compatibility with these enhancements, check each item in this note to be sure that you're following the recommendations.

If your software is written in a high-level language like Pascal or C and if you adhere to the guidelines listed in *Inside Macintosh*, many of the questions in this note won't concern you. If you develop in assembly language, you should read each question carefully. If you answer any question "yes," your software may encounter difficulty running on future Macintosh computers, and you should take the recommended action to change your software.

Do you depend on 68000 instructions which require that the processor be in supervisor mode?

In general, your software should not include instructions which depend on supervisor mode. These include modifying the contents of the status register. Most programs which modify the status register are only changing the Condition Code Register (CCR) half of the status register, so an instruction which addresses the CCR will work fine. Also, your software should not use the User Stack Pointer (USP) or turn interrupts on and off.

Do you have code which executes in response to an exception and relies on the position of data in the exception's local stack frame?

Exception stack frames vary on different microprocessors in the 68000 family, some of which may be used in future Macintosh computers. You should avoid using the `TRAP` instruction. **Note:** You can determine which microprocessor is installed by examining the low-memory global `CPUFlag` (a byte at `$12F`). These are the values:

CPUFlag	microprocessor
$00	68000
$01	68010
$02	68020
$03	68030

Do you use low-memory globals not documented in *Inside Macintosh*?

Other microprocessors in the 68000 family use the exception vectors in locations $0 through $FF in different ways. No undocumented location below the system heap ($100 through $13FF) is guaranteed to be available for use in future systems.

Do you make assumptions about the file system which are not consistent with both the original Macintosh File System and the Hierarchical File System?

Your applications should be compatible with both file systems. The easiest way to do this is to stick to the old files system trap calls (which work with both file systems) and avoid direct manipulation of data structures such as file control blocks and volume control blocks whenever possible.

Do you depend on the system or application heaps starting at a hard-coded address?

The starting addresses and the size of the system and application heaps has already changed (Macintosh vs. Macintosh Plus) and will change again in the future. Use the global `ApplZone` to find the application heap and `SysZone` to find the system heap. Also, don't count on the application heap zone starting at an address less than 65536 (that is, a system heap smaller than 64K).

Do you look through the system's queues directly?

In general, you should avoid examining queue elements directly. Instead, use the Operating System calls to manipulate queue elements.

Do you directly address memory-mapped hardware such as the VIA, the SCC, or the IWM?

You should avoid accessing this memory directly and use trap calls instead (disk driver, serial driver, etc.). Future machines may include a memory management unit (MMU) which may prevent access to memory-mapped hardware. Also, these memory-mapped devices may not be present on future machines. The addresses of these devices are likely to change, so if you must access the hardware directly, get the base address of the device from the appropriate low-memory global (obtainable from includes and interface files):

device	global
VIA	$1D4
SCCRd	$1D8
SCCWr	$1DC
IWM	$1E0

Do you assume the location or size of the screen?

The location, size, and bit depth of the screen is different in various machines. You can determine its location and size by examining the QuickDraw global variable `screenBits` on machines without Color QuickDraw. On machines with Color QuickDraw, the device list, described in the Graphics Devices chapter of *Inside Macintosh*, tells the location and size and bit depth of each screen, `screenBits` contains the location and size of the main device, and `GrayRgn` contains a region describing the shape and size of the desktop.

Does your software fail on some Macintosh models or on A/UX?

If so, you should determine the reason. Failure to run on all versions of the Macintosh may indicate problems which will prevent your software from working on future machines. Failture to run on A/UX, Apple's Unix for the Macintosh, also may indicate such problems.

Do you change master pointer flags of relocatable blocks directly with BSET or BCLR instructions?

In the future and on A/UX, all 32 bits of a master pointer may be used, with the flags byte moved elsewhere. Use the Memory Manager calls `HPurge`, `HNoPurge`, `HLock`, `HUnlock`, `HSetRBit`, `HClrRBit`, `HGetState`, and `HSetState` to manipulate the master pointer flags. (See the Memory Manager chapter of *Inside Macintosh Volume IV* for information on these calls.)

Do you check for 128K, 512K, and 1M RAM sizes?

You should be flexible enough to allow for non-standard memory sizes. This will allow your software to work in environments like MultiFinder.

Is your software incompatible with a third-party vendor's hardware?

If so, the incompatibility may prevent your software from working on future machines. You should research the incompatibility and try to determine a solution.

Do you rely on system resources being in RAM?

On most of our systems, some system resources are in ROM. You should not assume, for example, that you can regain RAM space by releasing system resources.

Does your software have timing-sensitive code?

Various Macintoshes run at different clock speeds, so timing loops will be invalid. You can use the trap call `Delay` for timing, or you can examine the global variable `Ticks`.

Do you have code which writes to addresses within the code itself?

A memory management unit (MMU) may one day prevent code from writing to addresses within code memory. Also, some microprocessors in the 68000 family cache code as it's encountered. Your data blocks should be allocated on the stack or in heap blocks separate from the code, and your code should not modify itself.

Do you rely on keyboard key codes rather than ASCII codes?

The various keyboards are slightly different; future keyboards may be different from them. For textual input, you should read ASCII codes rather than key codes.

Do you rely on the format of packed addresses in the trap dispatch table?

The trap dispatch table is different on various Macintoshes. There's no guarantee of the trap table's format in the future. You should use the system calls `GetTrapAddress` and `SetTrapAddress` to manipulate the trap dispatch table.

Do you use the Resource Manager calls `AddReference` or `RmveReference`?

These calls have been removed from the 128K ROM. They are no longer supported.

Do you store information in the application parameters area (the 32 bytes between the application and unit globals and the jump table)?

This space is reserved for use by Apple.

Do you depend on values in registers after a trap call, other than those documented in *Inside Macintosh*?

These values aren't guaranteed. The register conventions documented in *Inside Macintosh* will, of course, be supported. Often, you may not realize that your code is depending on these undocumented values, so check your register usage carefully.

Do you use the IMMED bit in File Manager calls?

This bit, which was documented in early versions of *Inside Macintosh* as a special form of File Manager call, actually did nothing for File Manager calls, and was used only for Device Manager calls. With the advent of the Hierarchical File System, this bit indicates that the call has a parameter block with hierarchical information.

Do you make assumptions about the number and size of disk drives?

There are now five sizes of Apple disks for the Macintosh (400K, 800K, and 20M, 40M, 80M), as well as many more from third-party vendors. You should use Standard File and File Manager calls to determine the number and size of disk drives.

Do you depend on alternate (page 2) sound or video buffers?

Some Macintoshes do not support alternate sound and video buffers.

Do you print by sending ASCII directly to the printer driver?

To retain compatibility with both locally-connected and AppleTalk-connected printers, you should print using Printing Managerr, as documented in *Inside Macintosh*.

Does your application fail when it's the startup application (i.e., without the Finder being run first)?

If so, you're probably not initializing a variable. If your application does not work as the startup application, you should determine why and fix the problem, since it may cause your application to fail in the future.

#3: Command-Shift-Number Keys

See also: The Toolbox Event Manager
Technical Note #110—MPW: Writing Standalone Code

Written by: Harvey Alcabes March 3,1985
Modified by: Ginger Jernigan April 25,1985
Updated: March 1, 1988

In the standard system, there are two Command-Shift-number key combinations that are automatically captured and processed by `GetNextEvent`. The combinations are:

Command-Shift-1	Eject internal disk
Command-Shift-2	Eject external disk

Numbers from 3 to 9 are also captured by `GetNextEvent`, but are processed by calling 'FKEY' resources. You can implement your own actions for Command-Shift-number combinations for numbers 5 to 9 by defining your own 'FKEY' resource. The routine must have no parameters. The ID of the resource must correspond to the number you want the routine to respond to. For example, if you want to define an action for Command-Shift-8, you would create an 'FKEY' resource with an ID of 8. The 'FKEY' resource should contain the code that you want to execute when the key is pressed.

The following Command-Shift-number key combinations are implemented with 'FKEY' resources in the standard System file.

Command-Shift-3	Save current screen as MacPaint file named Screen 0, Screen 1, ... Screen 9 (Works in one-bit mode only on Mac II)
Command-Shift-4	Print the active window (to an ImageWriter)
(with Caps Lock on)	Print the entire screen (to an ImageWriter)

#4: Error Returns from GetNewDialog

See also: The Dialog Manager

Written by: Russ Daniels
Updated:

April 4, 1985
March 1, 1988

When calling `GetNewDialog` to retrieve a dialog template from a previously opened resource file, how are error conditions indicated to the caller?

Unfortunately, they aren't. The Dialog Manager calls `GetResource` and assumes the returned value is good. Since the Dialog Manager doesn't check, you have two choices. Your first choice is to call `GetResource` for the dialog template, item list, and any resources needed by items in the item list yourself. But what do you do when you find the resources aren't there? Try to display an alert telling the user your application has been mortally wounded? What if resources needed for the alert aren't available?

The second, simpler alternative is to assure that the dialog template and other resources will be available when you build your product. This is really an adequate solution: If somebody uses a resource editor to remove your dialog template, you can hardly be blamed for its not executing properly.

A good debugging technique to catch this sort of problem is to put the value `$50FFC001` at absolute memory location 0 (the first long word of memory). If you do that, when the Dialog Manager tries to dereference the nil handle returned by the Resource Manager, you'll get an address error or bus error with some register containing `$50FFC001`. If you list the instructions around the program counter, you'll often see something like:

```
MOVE.L (A2),A1        ; in effect (0),A1
MOVE.L (A1),A1        ; the error occurs here
```

`GetNewWindow` and most of the other "GetSomething" calls will return nil if the "something" is not found.

#5: Using Modeless Dialogs from Desk Accessories

See also: The Toolbox Event Manager
The Dialog Manager
The Desk Manager

Written by: Russ Daniels April 4, 1985
Updated: March 1, 1988

When a desk accessory creates a window (including a modeless dialog window) it must set the windowKind to its refnum—a negative number. When the application calls `GetNextEvent`, the Event Manager calls `SystemEvent`, which checks to see if the event belongs to a desk accessory. `SystemEvent` checks the windowKind of the frontmost window, and uses the (negative) number for the refnum to make a control call, giving the desk accessory a shot at the event. Then `SystemEvent` returns TRUE, and `GetNextEvent` returns FALSE.

So, your desk accessory gets an event from `SystemEvent`. Since your window is a modeless dialog, you call `IsDialogEvent`, which mysteriously returns FALSE. What is going on?

Like `SystemEvent`, `IsDialogEvent` checks the windowKind of windows in the window list, looking for dialog windows. It does this by looking for windows with a windowKind of 2. In this case, it finds none, and does nothing.

The solution is to change the windowkind of your window to 2 before calling `IsDialogEvent`. This allows the Dialog Manager to recognize and handle the event properly. Before returning to `SystemEvent`, be sure to restore the windowKind. That way, when the application calls the Dialog Manager with the same event (the application should pass all events to Dialog Manager if it has any modeless dialogs itself), the Dialog Manager will ignore it.

#6: Shortcut for Owned Resources

See also: The Resource Manager
Technical Note #23—
Life With Font/DA Mover—Desk Accessories

Written by: Bryan Stearns May 10, 1986
Updated: March 1, 1988

To allow the Font/DA Mover to renumber desk accessories as needed when moving them between system files, desk accessories should use the "owned resource" protocol described in the Resource Manager chapter of *Inside Macintosh Volume I.*

All resource IDs in a desk accessory should be zero-based. At runtime, a routine can be called to find the current "base" value to add to a resource's zero-based value to get the actual current ID of that resource. Then, when a resource is needed, its zero-based value can be added to the resource base value, giving the actual resource ID to be used in future Resource Manager calls.

Here's the source to a handy routine to get the resource base value, `GetResBase`:

```
;FUNCTION GetResBase(driverNumber: INTEGER): INTEGER;
;
;GetResBase takes the driver number and returns the ID
;of the first resource owned by that driver. This is
;according to the private resource numbering convention
;documented in the Resource Manager.

GetResBase   FUNC

             MOVE.L     (SP)+,A0        ; Get return address
             MOVE.W     (SP)+,D0        ; Get driver number
             NOT.W      D0              ; Change to unit number
             ASL.W      #5,D0           ; Move it over in the word
             ORI.W      #$C000,D0       ; Add the magic bits
             MOVE.W     D0,(SP)         ; Return function result
             JMP        (A0)            ; and return

             END
```

#7: A Few Quick Debugging Tips

Written by: Jim Friedlander April 16, 1986
Updated: March 1, 1988

This presents a few tips which may make your debugging easier.

Setting memory location 0 to something odd

Dereferencing nil handles can cause real problems for an application. If location 0 (nil) is something even, the dereference will not cause an address error, and the application can run on for quite a while, making tracing back to the problem quite difficult. If location 0 contains something odd, such as `$50FFC001`, an address error will be generated immediately when a nil handle is dereferenced. On Macintoshes with 68020s, like the Mac II, this same value (`$50FFC001`) will cause a bus error. An address error or bus error will also be generated, of course, when the ROM tries to dereference a nil handle, such as when you call `HNoPurge(hndl)`, where `hndl` is nil.

Some versions of the TMON debugger set location 0 to 'NIL!' (`$4E494C21`) or `$50FFC001`. If you are using MacsBug, you should include code in your program that sets location 0. Of course, there is no need to ship your application with this code in it—it's just for debugging purposes. Old versions of the Finder used to set location 0 to the value $464F424A ('FOBJ'). On newer machines, newly launched applications get location 0 set to `$00F80000` by the Segment Loader.

Checksumming for slow motion mode

Entering the Macsbug command "`SS 400000 400000`" will cause Macsbug to do a checksum of the location `$400000` every time an instruction is executed. Checksum the ROM, because it will not change while your program is executing (the ROM may change in between launches of your application, but that's OK)! This will cause the Macintosh to go into slow motion mode. For example, you will need to hold down the mouse button for about 10 seconds to get a menu to pull down—you can see how the ROM draws menus, grays text, etc.

This technique is very handy for catching problems like multiple updates of your windows, redrawing scroll bars more than once, that troublesome flashing grow icon, etc. To turn slow motion mode off, simply enter MacsBug and type "`SS`".

TMON performs this function in a different way. Instead of calculating the checksum after each instruction, it only calculates checksums after each trap. You can checksum different amounts of the ROM depending on how much you want things to slow down.

Checksumming MemErr

A lot of programs don't call `MemError` as often as they should. If you are having strange, memory-related problems, one thing that you can do to help find them is to checksum on `MemErr` (the low memory global word at `$220`). In MacsBug, type "`SS 220 221`". In TMON, enter 220 and 221 as limits on the '`Checksum (bgn end) :`' line and on the line above, enter the range of traps you wish to have the checksum calculated after.
When `MemErr` changes, the debugger will appear, and you can check your code to make sure that you are checking `MemErr`. If not, you might have found a problem that could cause your program to crash!

Checksumming on a master pointer

Due to fear of moving memory, some programmers lock every handle that they create. Of course, handles need only be locked if they are going to be dereferenced **and** if a call will be made that can cause relocation. Unnecessarily locking handles can cause unwanted heap fragmentation. If you suspect that a particular memory block is moving on you when you have its handle dereferenced, you can checksum the master pointer (the handle you got back from `NewHandle` is the address of the master pointer). Your program will drop into the debugger each time your handle changes—that is, either when the block it refers to is relocated, or when the master pointer's flags byte changes.

#8: RecoverHandle Bug in AppleTalk Pascal Interfaces

See also: AppleTalk Manager

Written by: Bryan Stearns
Updated:

April 21, 1986
March 1, 1988

Previous versions of this note described a bug in the AppleTalk Pascal Interfaces. This bug was fixed in MPW 1.0 and newer.

Macintosh
Technical Notes



Developer Technical Support

#9: Will Your AppleTalk Application Support Internets?

Written by: Sriram Subramanian & Pete Helme
Written by: Bryan Stearns

April 1990
April 1986

This Technical Note discusses how AppleTalk applications should work across internets, groups of interconnected AppleTalk networks. It explains the differences between life on a single AppleTalk network and life on an internet.
Changes since March 1988: Removed the section on AppleTalk retry timers, as it is no longer accurate; see Technical Note #270, AppleTalk Timers Explained, for more information on retry timers.

You can read about internets (AppleTalk networks connect by one or more bridges) in *Inside AppleTalk*. What do you need to do about them?

Use a High-Level Network Protocol

Make sure you use the Datagram Delivery Protocol (DDP), or a higher AppleTalk protocol based on DDP, like the AppleTalk Transaction Protocol (ATP). Be warned that Link Access Protocol (LAP) packets do not make it across bridges to other AppleTalk networks. Also, don't broadcast; broadcast packets are not forwarded by bridges (broadcasting using protocols above LAP is discouraged, anyway).

Use Name Binding

As usual, use the Name Binding Protocol (NBP) to announce your presence on the network, as well as to find other entities on the network. Pay special attention to zone name fields; the asterisk (as in "MyLaser:LaserWriter:*") in a name you look up is now important; it means "my zone only" (see the Zone Information Protocol (ZIP) chapter of *Inside AppleTalk* for information on finding out what other zones exist). The zone field should always be an asterisk when registering a name.

Pay Attention to Network Number Fields

When handling the network addresses returned by `NBPLookUp` (or anyone else), don't be surprised if the network number field is non-zero.

Am I Running on an Internet?

The low-memory global `ABridge` is used to keep track of a bridge on the local AppleTalk network (NBP and DDP use this value). If `ABridge` is non-zero, then you're running on an internet; if it's zero, chances are, you're not (this is not guaranteed, however, due to the fact that the `ABridge` value is "aged", and if NBP hasn't heard from the bridge in a long time, the value is cleared).

Watch for Out-Of-Sequence and Non-Exactly-Once Requests

Due to a "race" condition on an internet, it's possible for an exactly-once ATP packet to slip through twice; to keep this from happening, send a sequence number as part of the data with each ATP packet; whenever you make a request, bump the sequence number, and never honor an old sequence number.

Further Reference:

- *Inside AppleTalk*
- *Inside Macintosh*, Volumes II & V, The AppleTalk Manager
- Technical Note #250, AppleTalk Phase 2 on the Macintosh
- Technical Note #270, AppleTalk Timers Explained

Macintosh Technical Notes

#10: Pinouts

See also: *Macintosh Hardware Reference Manual*
Technical Note #65—Macintosh Plus Pinouts

Written by: Mark Baumwell April 26, 1985
Modified: July 23, 1985
Updated: March 1, 1988

This note gives pinouts for Macintosh ports, cables, and other products.

Below are pinout descriptions for the Macintosh ports, cables, and various other products. Please refer to the Hardware chapter of *Inside Macintosh* and the *Macintosh Hardware Reference Manual* for more information, especially about power limits. Note that unconnected pins are omitted.

Macintosh Port Pinouts

Macintosh Serial Connectors (DB-9)

Pin	Name	Description/Notes
1	Ground	
2	+5V	See *Inside Macintosh* for power limits
3	Ground	
4	TxD+	Transmit Data line
5	TxD–	Transmit Data line
6	+12V	See Macintosh Hardware chapter for power limits
7	HSK	HandShaKe: CTS or TRxC, depends on Zilog 8530 mode
8	RxD+	Receive Data line; ground this line to emulate RS232
9	RxD–	Receive Data line

Macintosh Mouse Connector (DB-9)

Pin	Name	Description/Notes
1	Ground	
2	+5V	See *Inside Macintosh* for power limits
3	GND	Ground
4	X2	Horizontal movement line (connected to VIA PB4 line)
5	X1	Horizontal movement line (connected to SCC DCDA– line)
7	SW–	Mouse button line (connected to VIA PB3)
8	Y2	Vertical movement line (connected to VIA PB5 line)
9	Y1	Vertical movement line (connected to SCC DCDB– line)

Macintosh Keyboard Connector (RJ-11 Telephone-style jack)

Pin	Name	Description/Notes
1	Ground	
2	KBD1	Keyboard clock
3	KBD2	Keyboard data
4	+5V	See *Inside Macintosh* for power limits

Macintosh External Drive Connector (DB-19)

Pin	Name	Description/Notes
1	Ground	
2	Ground	
3	Ground	
4	Ground	
5	–12V	See *Inside Macintosh* for power limits
6	+5V	See *Inside Macintosh* for power limits
7	+12V	See *Inside Macintosh* for power limits
8	+12V	See *Inside Macintosh* for power limits
10	PWM	Regulates speed of the drive
11	PH0	Control line to send commands to the drive
12	PH1	Control line to send commands to the drive
13	PH2	Control line to send commands to the drive
14	PH3	Control line to send commands to the drive
15	WrReq–	Turns on the ability to write data to the drive
16	HdSel	Control line to send commands to the drive
17	Enbl2–	Enables the Rd line (else Rd is tri-stated)
18	Rd	Data actually read from the drive
19	Wr	Data actually written to the drive

Other Pinouts

Macintosh XL Serial Connector A (DB-25)

Pin	Name	Description/Notes
1	Ground	
2	TxD	Transmit Data line
3	RxD	Receive Data line
4	RTS	Request to Send
5	CTS	Clear To Send
6	DSR	Data Set Ready
7	Ground	
8	DCD	Data Carrier Detect
15	TxC	Connected to TRxCA
17	RxC	Connected to RTxCA
24	TEXT	Connected to TRxCA

Macintosh XL Serial Connector B (DB-25)

Pin	Name	Description/Notes
1	Ground	
2	TxD–	Transmit Data line
3	RxD–	Receive Data line
6	HSK/DSR	TRxCB or CTSB
7	Ground	
19	RxD+	Receive Data line
20	TXD+/DTR	connected to DTRB

Apple 300/1200 Modem Serial Connector (DB-9)

Modem	Name	Description/Notes
2	DSR	Output from modem
3	Ground	
5	RxD	Output from modem
6	DTR	Input to modem
7	DCD	Output from modem
8	Ground	
9	TxD	Input to modem

Apple ImageWriter Serial Connector (DB-25)

ImageWriter	Name	Description/Notes
1	Ground	
2	SD	Send Data; Output from ImageWriter
3	RD	Receive Data; Input to ImageWriter
4	RTS	Output from ImageWriter
7	Ground	
14	FAULT–	False when deselected; Output from ImageWriter
20	DTR	Output from ImageWriter

Apple LaserWriter AppleTalk Connector (DB-9)

LaserWriter	Name	Description/Notes
1	Ground	
3	Ground	
4	TxD+	Transmit Data line
5	TxD–	Transmit Data line
7	RXCLK	TRxC of Zilog 8530
8	RxD+	Receive Data line
9	RxD–	Receive Data line

Apple LaserWriter Serial Connector (DB-25)

LaserWriter	Name	Description/Notes
1	Ground	
2	TXD–	Transmit Data; Output from LaserWriter
3	RXD–	Receive Data; Input to LaserWriter
4	RTS–	Output from LaserWriter
5	CTS	Input to LaserWriter
6	DSR	Input to LaserWriter (connected to DCBB– of 8530)
7	Ground	
8	DCD	Input to LaserWriter (connected to DCBA– of 8530)
20	DTR–	Output from LaserWriter
22	RING	Input to LaserWriter

Macintosh Cable Pinouts

Note for the cable descriptions below:

The arrows ("→") show which side is an input and which is an output. For example, the notation "a → b" means that signal "a" is an output and "b" is an input.

When pins are said to be connected on a side in the Notes column, it means the pins are connected on that side of the connector.

Macintosh ImageWriter Cable
(part number 590-0169)

Macintosh (DB9)	Name			ImageWriter (DB25)	Notes
1	Ground			1	
3	Ground			7	pins 3, 8 connected on Macintosh side
5	TxD–	→	RD	3	RD = Receive Data
7	HSK	←	DTR	20	
8	RxD+	=	GND		Not connected on ImageWriter side
9	RxD–	←	SD	2	SD = Send Data

Macintosh Modem Cable (Warning! Don't use this cable to connect 2 Macintoshes!)
(part number 590-0197-A)

Macintosh (DB9)	Name			Modem (DB9)	Notes
3	Ground			3	pins 3, 8 connected on EACH side
5	TxD–	→	TxD	9	
6	+12V	→	DTR	6	
7	HSK	←	DCD	7	
8	No wire			8	
9	RxD–	←	RxD	5	

Macintosh to Macintosh Cable (Macintosh Modem Cable with pin 6 clipped on both ends.)

Macintosh (DB9)	Name			Macintosh (DB9)	Notes
3	Ground			3	pins 3, 8 connected on EACH side
5	TxD–	→	RxD–	9	
7	HSK	←	DCD	7	
8	No wire			8	
9	RxD–	←	TxD–	5	

Macintosh External Drive Cable (part number 590-0183-B)

Macintosh (DB9)	Name	Sony Drive (20 Pin Ribbon)
1	Ground	1
2	Ground	3
3	Ground	5
4	Ground	7
6	+5V	11
7	+12V	13
8	+12V	15
10	PWM	20
11	PH0	2
12	PH1	4
13	PH2	6
14	PH3	8
15	WrReq–	10
16	HdSel	12
17	Enbl2–	14
18	Rd	16
19	Wr	18

Macintosh XL Null Modem Cable (part number 590-0166-A)

Macintosh XL (DB25)	Name			DTE (DB25)	Notes
1	Ground			1	
2	TxD–	→	RxD	3	
3	RxD–	←	TxD	2	
4, 5	RTS,CTS	→	DCD	8	pins 4, 5 connected together
6	DSR	←	DTR	20	
7	Ground			7	
8	DCD	←	RTS, CTS	4, 5	pins 4, 5 connected together
20	DTR	→	DSR	6	

Macintosh to Non-Apple Product Cable Pinouts

Macintosh to IBM PC Serial Cable #1 (not tested)

Macintosh (DB9)	Name			IBM PC (DB25)	Notes
3	Ground			7	pins 3, 8 connected on Macintosh side
5	TxD–	→	RxD	3	
7	HSK	←	DTR	20	
8	RxD+	=	Ground		Not connected on IBM side
9	RxD–	←	TxD	2	
	CTS	←	RTS	4-5	pins 4, 5 connected on IBM side
	DSR	←	DCD,DTR	6-8-20	pins 6, 8, 20 connected on IBM side

Macintosh to IBM PC Serial Cable #2 (not tested)

Macintosh (DB9)	Name			IBM PC (DB25)	Notes
1	Ground			1	
3	Ground			7	pins 3, 8 connected on Macintosh side
5	TxD–	→	RxD	3	
9	RxD–	←	TxD	2	
	CTS	←	RTS	4-5	pins 4, 5 connected on IBM side
	DSR	←	DTR	6-8	pins 6, 8 connected on IBM side



#11: Memory–Based MacWrite Format

Revised: August 1989

This Technical Note formerly described the format of files created by MacWrite® 2.2.
Changes since March 1988: Updated the CLARIS address.

This Note formerly discussed the memory–based MacWrite 2.2 file format. For information on MacWrite and other CLARIS products, contact CLARIS at:

CLARIS Corporation
5201 Patrick Henry Drive
P.O. Box 58168
Santa Clara, CA 95052-8168

Technical Support
Telephone: (408) 727-9054
AppleLink: Claris.Tech

Customer Relations
Telephone: (408) 727-8227
AppleLink: Claris.CR

MacWrite is a registered trademark of CLARIS Corporation.



#12: Disk–Based MacWrite Format

Revised: August 1989

This Technical Note formerly described the format of files created by MacWrite®, which is now published by CLARIS.
Changes since March 1988: Updated the CLARIS address.

This Note formerly discussed the disk–based MacWrite file format. For information on MacWrite and other CLARIS products, contact CLARIS at:

CLARIS Corporation
5201 Patrick Henry Drive
P.O. Box 58168
Santa Clara, CA 95052-8168

Technical Support
Telephone: (408) 727-9054
AppleLink: Claris.Tech

Customer Relations
Telephone: (408) 727-8227
AppleLink: Claris.CR

MacWrite is a registered trademark of CLARIS Corporation.



#13: MacWrite Clipboard Format

Revised: August 1989

This Technical Note formerly described the clipboard format used by MacWrite®, which is now published by CLARIS.
Changes since March 1988: Updated the CLARIS address.

This Note formerly discussed the MacWrite clipboard format. For information on MacWrite and other CLARIS products, contact CLARIS at:

CLARIS Corporation
5201 Patrick Henry Drive
P.O. Box 58168
Santa Clara, CA 95052-8168

Technical Support
Telephone: (408) 727-9054
AppleLink: Claris.Tech

Customer Relations
Telephone: (408) 727-8227
AppleLink: Claris.CR

MacWrite is a registered trademark of CLARIS Corporation.

#14: The INIT 31 Mechanism

See: The System Resource File
The Start Manager

Written by: Bryan Stearns March 13, 1986
Updated: March 1, 1988

This note formerly described things that are now covered in the System Resource File chapter of *Inside Macintosh Volume IV* and the Start Manager chapter of *Inside Macintosh Volume V*. Please refer to *Inside Macintosh*.

#15: Finder 4.1

Written by: Harvey Alcabes
Updated:

April 12, 1985
March 1, 1988

This note formerly described Finder 4.1, which is now recommended only for use with 64K ROM machines. Information specific to 64K ROM machines has been deleted from Macintosh Technical Notes for reasons of clarity.

#16: MacWorks XL

Written by: Harvey Alcabes
Mark Baumwell

May 11, 1985

Updated: March 1, 1988

Earlier versions of this note described MacWorks XL, the system software that allowed you to use Macintosh applications on the Macintosh XL. Information specific to Macintosh XL machines has been deleted from Macintosh Technical Notes for reasons of clarity.

#17: Low-Level Print Driver Calls

See also: The Print Manager

Written by: Ginger Jernigan April 14, 1986
Updated: March 1, 1988

This technical note has been replaced by information in *Inside Macintosh Volume V*. Please refer to the Print Manager chapter of *Inside Macintosh Volume V* for information on low-level print driver calls.

#18: TextEdit Conversion Utility

See also: Macintosh Memory Management: An Introduction
TextEdit

Written by: Harvey Alcabes
Updated:

April 10, 1985
March 1, 1988

Text sometimes must be converted between a Pascal string and "pure" text in a handle. This note illustrates a way to do this using MPW Pascal.

Text contained in TextEdit records sometimes must be passed to routines which expect a Pascal string of type Str255 (a length byte followed by up to 255 characters). The following MPW Pascal unit can be used to convert between TextEdit records and Pascal strings:

```
UNIT TEConvert;

   {General utilities for conversion between TextEdit and strings}

   INTERFACE

      USES MemTypes,QuickDraw,OSIntf,ToolIntf;

      PROCEDURE TERecToStr(hTE: TEHandle; VAR str: Str255);
      {TERecToStr converts the TextEdit record hTE to the string str.}
      {If necessary, the text will be truncated to 255 characters.}

      PROCEDURE StrToTERec(str: Str255; hTE: TEHandle);
      {StrToTERec converts the string str to the TextEdit record hTE. }

   IMPLEMENTATION

      PROCEDURE TERecToStr(hTE: TEHandle; VAR str: Str255);

         BEGIN
            GetIText(hTE^^.hText, str);
         END;

      PROCEDURE StrToTERec(str: Str255; hTE: TEHandle);

         BEGIN
            TESetText(POINTER(ORD4(@str) + 1), ORD4(length(str)), hTE);
         END;

END.
```

Macintosh
Technical Notes



Developer Technical Support

#19: How To Produce Continuous Sound Without Clicking

Revised by: Jim Reekes June 1989
Written by: Ginger Jernigan April 1985

This Technical Note formerly described how to use the Sound Driver to produce continuous sound without clicking.
Changes since March 1988: The continuous sound technique is no longer recommended.

Apple currently discourages use of the Sound Driver due to compatibility issues. The hardware support for sound designed into the early Macintosh architecture was minimal. (Many things have changed since 1983–1984.) The new Macintosh computers contain a custom chip to provide better support for sound, namely the Apple Sound Chip (ASC). The ASC is present in the complete Macintosh II family as well as the Macintosh SE/30 and later machines. When the older hardware of the Macintosh Plus and SE are accessed, it is likely to cause a click. This click is a hardware problem. The software solution to this problem was to continuously play silence. This is not a real solution to the problem and is not advisable for the following reasons:

- The Sound Driver is no longer supported. There have always been, and still are, bugs in the glue code for `StartSound`.
- The Sound Driver may not be present in future System Software releases, or future hardware may not be able to support it. The Sound Manager is the application's interface to the sound hardware.
- The technique used to create a continuous sound should have only been used on a Macintosh Plus or SE, since these are the only models that have the "embarrassing click." Do not use this method on a Macintosh which has the Apple Sound Chip.
- Using the continuous sound technique, or the Sound Driver for that matter, will cause problems for the system and those applications that properly use the Sound Manager. Also realize that `_SysBeep`, which is a common routine that everything uses, is a Sound Manager routine.
- The continuous sound technique wastes CPU time by playing silence. With multimedia applications and the advent of MultiFinder, it is important to allow the CPU to do as much work as possible. The continuous sound technique used the CPU to continuously play silence, thus stealing valuable time from other, more important, jobs.

Further Reference:

- *The Sound Manager,* Interim Chapter by Jim Reekes, October 2, 1988
- Technical Note #180, MultiFinder Miscellanea

#20: Data Servers on AppleTalk

See also: The AppleTalk Manager
Inside LaserWriter

Written by: Bryan Stearns April 29, 1985
Updated: March 1, 1988

Many applications could benefit from the ability to share common data between several Macintoshes, without requiring a file server. This technical note discusses one technique for managing this AppleTalk communication.

There are four main classes of network "server" devices:

Device Servers, such as the LaserWriter, allow several users to share a single hardware device; other examples of this (currently under development by third parties) are modem servers and serial servers (to take advantage of non-intelligent printers such as the ImageWriter).

File Servers, such as AppleShare, which support file access operations over the network. A user station sends high-level requests over the network (such as "Open this file," "Read 137 bytes starting at the current file position of this file," "Close this file," etc.).

Block Servers, which answer to block requests over the network. These requests impart no file system knowledge about the blocks being passed, i.e., the server doesn't know which files are open by which users, and therefore cannot protect one user's open file from other users. Examples of this type of server are available from third-party developers.

Data Servers, which answer to requests at a higher level than file servers, such as "Give me the first four records from the database which match the following search specification." This note directs its attention at this type of server.

A data server is like a file server in that it responds to intelligent requests, but the requests that it responds to can be more specialized, because the code in the server was written to handle that specific type of request. This has several added benefits: user station processing can be reduced, if the data server is used for sorting or searching operations; and network traffic is reduced, because of the specificity of the requests passed over the network. The data server can even be designed to do printing (over the network to a LaserWriter, or on a local ImageWriter), given that it has the data and can be directed as to the format in which it should be printed.

ATP: The AppleTalk Transaction Protocol

ATP, the assured-delivery AppleTalk Transaction Protocol, can be used to support all types of server communications (the LaserWriter uses ATP for its communications!). Here is a possible scenario between two user stations ("Dave" and "Bill") and a data server station ("OneServer", a server of type "MyServer"). We've found that the "conversational" analogy is helpful when planning AppleTalk communications; this example is therefore presented as a conversation, along with appropriate AppleTalk Manager calls (Note that no error handling is presented, however; your application should contain code for handling errors, specifically the "half-open connection" problem described below).

Establishing the Connection

Each station uses `ATPLoad` to make sure that AppleTalk is loaded. The server station, since it wants to accept requests, opens a socket and registers its name using `NBPRegister`. The user stations use `NBPLookUp` to find out the server's network address. This looks like this, conversationally:

Server:	"I'm ready to accept requests!"	`ATPLoad`
		`OpenSocket`
		`NBPRegister`
		`ATPGetRequest`
		`ATPGetRequest`
		`ATPGetRequest`
			Opens AppleTalk
			Creates socket
			Assigns name to socket
			queue a few asynchronous calls, to be able to handle several users

Dave:	"Any 'MyServers' out there?"	`ATPLoad` `NBPLookup`	Opens AppleTalk look for servers, finds OneServer
Dave:	"Hey, MyServer! What socket should I talk to you on?"	`ATPRequest`	Ask the server which socket to use for further communications
Bill:	"Any 'MyServers' out there?"	`ATPLoad` `NBPLookup`	Opens AppleTalk look for servers, finds OneServer
Bill:	"Hey, MyServer! What socket should I talk to you on?"	`ATPRequest`	Ask the server which socket to use for further communications

Server:	"Hi, Dave! Use Socket N."	`ATPOpenSkt`
		`ATPResponse`
		`ATPGetRequest`
			Get a new socket for talking to Dave
			Send Dave the socket number
			Replace the used GetRequest

Server:	"Hi, Bill! Use socket M."	`ATPOpenSkt`
		`ATPResponse`
		`ATPGetRequest`
			Get a new socket for talking to Bill
			Send Bill the socket number
			Replace the used GetRequest

From this point on, the server knows that any requests received on socket N are from Dave, and those received on socket M are from Bill. The conversations continue, after a brief discussion of error handling.

Half-Open Connections

There is a possibility that one side of a connection could go down (be powered off, rebooted accidently, or simply crash) before the connection has been officially broken. If a user station goes down, the server must throw away any saved state information and close that user's open socket. This can be done by requiring that the user stations periodically "tickle" the server: every 30 seconds (for example) the user station sends a dummy request to the server, which sends a dummy response. This lets each side of the connection know that the other side is still "alive."

When the server detects that two intervals have gone by without a tickle request, it can assume that the user station has crashed, and close that user's socket and throw away any accumulated state information.

The user station should use a vertical-blanking task to generate these tickle requests asyncronously, rather than generating them within the `GetNextEvent` loop; this avoids problems with long periods away from `GetNextEvent` (such as when a modal dialog box is running). This task can look at the time that the last request was made of the server, and if it's approaching the interval time, queue an **asynchronous** request to tickle the server (it's important that any AppleTalk calls made from interrupt or completion routines be asynchronous).

If a user station's request (including a tickle request) goes unanswered, the user station should recover by looking for the server and reestablishing communications as shown above (beginning with the call to `NBPLookUp`).

More information about half-open connections can be found in the Printer Access Protocol chapter of *Inside LaserWriter,* available from APDA.

Using the Connection

The user stations Dave and Bill have established communications with the server, each on its own socket (note that the user stations have not had to open their own sockets, or register names of their own, to do this—the names we're using are merely for explanational convenience). They are also automatically tickling the server as necessary.

Now the user stations make requests of the server as needed:

Bill:	"I'd like to use the sales figures for this year."	`ATPRequest`	Bill opens a database.
Server:	"Ok, Bill."	`ATPResponse`	The server checks to make sure that no one else is using that database.
Dave:	"Hey, Server - I'm still here!"	`ATPRequest`	Dave notices that the interval time is approaching, and makes a tickle request.
Server:	"Ok, Dave."	`ATPResponse`	The server resets its "last time I heard from Dave".
Bill:	"Please print the figures for January thru June."	`ATPRequest`	Bill asks for specific data.
Server:	"Ok, Bill."	`ATPResponse`	The server does a database search sorts the results, and prints them on a local Imagewriter.
Dave:	"I'd like to use the sales figures for this year."	`ATPRequest`	Dave opens a database.
Server:	"Sorry, Dave, I can't do that. Bill is using that database."	`ATPResponse`	The server finds that Bill is using that data.

Closing the Connection

The user stations continue making requests of the server, until each is finished. The type of work being done by the server determines how long the conversation will last: since the number of sockets openable by the server is limited, it may be desirable to structure the requests in such a way that the average conversation is very short. It may also be necessary to have a (NBP named) socket open on the user station, if the server needs to communicate with the user on other than a request-response basis. Here is how our example connections ended:

Dave:	"Thank you, server, I'm done now. You've been a big help."	`ATPRequest`	Dave tells the server he's finished.

Server:	"Ok, Dave. Bye now."	`ATPResponse`
		`ATPCloseSkt`
		
		`ATPCloseSkt`
			The server kisses Dave goodbye. After the Response operation completes, the server closes the socket Dave was using. It also notices that Bill hasn't sent a request in more than two intervals, and closes Bill's socket, too.

The user station can forget about the socket it was using on the server; if it needs to talk with the server again, it starts at the `NBPLookUp` (just in case the server has moved, gone down and come up, etc.).

#21: QuickDraw's Internal Picture Definition

See also: QuickDraw
Color QuickDraw
Using Assembly Language
Technical Note #59—Pictures and Clip Regions

Written by: Ginger Jernigan — April 24, 1985
Modified by: Rick Blair — November 15, 1986
Updated: March 1, 1988

This technical note describes the internal format of the QuickDraw picture data structure. This revision corrects some errors in the opcode descriptions and provides some examples.

This technical note describes the internal definition of the QuickDraw picture. The information given here only applies to QuickDraw picture format version 1.0 (which is always created by Macintoshes without Color QuickDraw). Picture format version 2.0 is documented in the Color QuickDraw chapter of *Inside Macintosh*. This information should not be used to write your own picture bottleneck procedures; if we add new objects to the picture definition, your program will not be able to operate on pictures created using standard QuickDraw. Your program will not know the size of the new objects and will, therefore, not be able to proceed past the new objects. (What this ultimately means is that you can't process a new picture with an old bottleneck proc.)

Terms

An *opcode* is a number that `DrawPicture` uses to determine what object to draw or what *mode* to change for subsequent drawing. The following list gives the opcode, the name of the object (or mode), the associated data, and the total size of the opcode and data. To better interpret the sizes, please refer to page I-91 of the Using Assembly Language chapter of *Inside Macintosh*. For types not described there, here is a quick list:

opcode	byte
mode	word
point	4 bytes
0..255	byte
–128..127	signed byte
rect	8 bytes
poly	10+ bytes (starts with word size for poly (incl. size word)
region	10+ bytes (starts with word size for region (incl. size word)

fixed point	long
pattern	8 bytes
rowbytes	word (always even)
bit data	rowbytes * (bounds.bottom - bounds.top) bytes

Each picture definition begins with a `picsize` (`word`), then a `picframe` (`rect`), and then the picture definition, which consists of a combination of the following opcodes:

Opcode	Name	Additional Data	Total Size (bytes)
00	NOP	none	1
01	clipRgn	rgn	1+rgn
02	bkPat	pattern	9
03	txFont	font (word)	3
04	txFace	face (byte)	2
05	txMode	mode (word)	3
06	spExtra	extra (fixed point)	5
07	pnSize	pnSize (point)	5
08	pnMode	mode (word)	3
09	pnPat	pattern	9
0A	thePat	pattern	9
0B	ovSize	point	5
0C	origin	dh, dv (words)	5
0D	txSize	size (word)	3
0E	fgColor	color (long)	5
0F	bkColor	color (long)	5
10	txRatio	numer (point), denom (point)	9
11	picVersion	version (byte)	2
20	line	pnLoc (point), newPt (point)	9
21	line from	newPt (point)	5
22	short line	pnLoc (point); dh, dv (-128..127)	7
23	short line from	dh, dv (-128..127)	3
28	long text	txLoc (point), count (0..255), text	6+text
29	DH text	dh (0..255), count (0..255), text	3+text
2A	DV text	dv (0..255), count (0..255), text	3+text
2B	DHDV text	dh, dv (0..255), count (0..255), text	4+text
30	frameRect	rect	9
31	paintRect	rect	9
32	eraseRect	rect	9
33	invertRect	rect	9
34	fillRect	rect	9
38	frameSameRect	rect	1
39	paintSameRect	rect	1
3A	eraseSameRect	rect	1
3B	invertSameRect	rect	1
3C	fillSameRect	rect	1
40	frameRRect	rect (ovalwidth, height; see 1, below)	9
41	paintRRect	rect (ovalwidth, height; see 1, below)	9
42	eraseRRect	rect (ovalwidth, height; see 1, below)	9

Opcode (cont.)	Name	Additional Data	Total Size (bytes)
43	invertRRect	rect (ovalwidth, height; see 1, below)	9
44	fillRRect	rect (ovalwidth, height; see 1, below)	9
48	frameSameRRect	rect	1
49	paintSameRRect	rect	1
4A	eraseSameRRect	rect	1
4B	invertSameRRect	rect	1
4C	fillSameRRect	rect	1
50	frameOval	rect	9
51	paintOval	rect	9
52	eraseOval	rect	9
53	invertOval	rect	9
54	fillOval	rect	9
58	frameSameOval	rect	1
59	paintSameOval	rect	1
5A	eraseSameOval	rect	1
5B	invertSameOval	rect	1
5C	fillSameOval	rect	1
60	frameArc	rect, startAngle, arcAngle	13
61	paintArc	rect, startAngle, arcAngle	13
62	eraseArc	rect, startAngle, arcAngle	13
63	invertArc	rect, startAngle, arcAngle	13
64	fillArc	rect, startAngle, arcAngle	13
68	frameSameArc	startAngle, arcAngle	5
69	paintSameArc	startAngle, arcAngle	5
6A	eraseSameArc	startAngle, arcAngle	5
6B	invertSameArc	startAngle, arcAngle	5
6C	fillSameArc	startAngle, arcAngle	5
70	framePoly	poly	1+poly
71	paintPoly	poly	1+poly
72	erasePoly	poly	1+poly
73	invertPoly	poly	1+poly
74	fillPoly	poly	1+poly
78	frameSamePoly	(not yet implemented—same as 70, etc.)	1
79	paintSamePoly	(not yet implemented)	1
7A	eraseSamePoly	(not yet implemented)	1
7B	invertSamePoly	(not yet implemented)	1
7C	fillSamePoly	(not yet implemented)	1
80	frameRgn	rgn	1+rgn
81	paintRgn	rgn	1+rgn
82	eraseRgn	rgn	1+rgn
83	invertRgn	rgn	1+rgn
84	fillRgn	rgn	1+rgn
88	frameSameRgn	(not yet implemented—same as 80, etc.)	1
89	paintSameRgn	(not yet implemented)	1
8A	eraseSameRgn	(not yet implemented)	1
8B	invertSameRgn	(not yet implemented)	1

Opcode (cont.)	Name	Additional Data	TotalSize (bytes)
8C	fillSameRgn	(not yet implemented)	1
90	BitsRect	rowBytes, bounds, srcRect, dstRect, mode, unpacked bitData	29+unpacked bitData
91	BitsRgn	rowBytes, bounds, srcRect, dstRect, mode, maskRgn, unpacked bitData	29+rgn+ bitData
98	PackBitsRect	rowBytes, bounds, srcRect, dstRect, mode, packed bitData for each row	29+packed bitData
99	PackBitsRgn	rowBytes, bounds, srcRect, dstRect, mode, maskRgn, packed bitData for each row	29+rgn+ packed bitData
A0	shortComment	kind(word)	3
A1	longComment	kind(word), size(word), data	5+data
FF	EndOfPicture	none	1

Notes

Rounded-corner rectangles use the setting of the ovSize point (see opcode $0B, above).

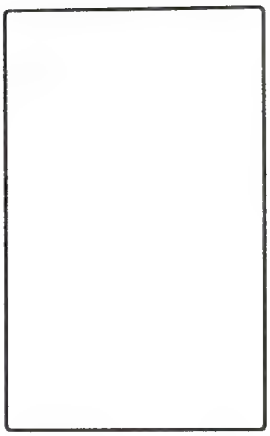
`OpenPicture` and `DrawPicture` set up a default set of port characteristics when they start. When drawing occurs, if the user's settings don't match the defaults, mode opcodes are generated. This is why there is usually a `clipRgn` code after the `picVersion`: the default clip region is an empty rectangle.

The only savings that the "same" opcodes achieve under the current implementation is for rectangles. `DrawPicture` keeps track of the last rectangle used and if a "same" opcode is encountered that requests a rectangle, the last rect. will be used (and no rectangle will appear in the opcode's data).

This last section contains some Pascal program fragments that generate pictures. Each section starts out with the picture itself (yes, they're dull) followed by the code to create and draw it, and concludes with a commented hex dump of the picture.

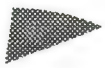
```
{variables used in all examples}

VAR
    err:     OSErr;
    ph:      PicHandle;
    h:       Handle;
    r:       Rect;
    smallr:  Rect;
    orgr:    Rect;
    pstate:  PenState; {are they in the Rose Bowl, or the state pen?}
```



```
I.    {Rounded-corner rectangle}
      SetRect(r, 20, 10, 120, 175);
      ClipRect(myWindow^.portRect);
      ph := OpenPicture(r);
      FrameRoundRect (r, 5, 4); {r,width,height}
      ClosePicture;
      DrawPicture(ph, r);

'PICT' (1)  0026 {size}  000A 0014 00AF 0078 {picFrame}
      1101 {version 1}  01 000A 0000 0000 00FA 0190 {clipRgn - 10 byte region}
      0B 0004 0005 {ovSize point}  40 000A 0014 00AF 0078 {frameRRect rectangle}
      FF {fin}
```



```
II.   {Overpainted arc}
      GetPenState(pstate); {save}
      SetRect(r, 20, 10, 120, 175);
      ClipRect(myWindow^.portRect);
      ph := OpenPicture(r);
      PaintArc(r, 3, 45); {r,startangle,endangle}
      PenPat(gray);
      PenMode(patXor); {turn the black to gray}
      PaintArc(r, 3, 45); {r,startangle,endangle}
      ClosePicture;
      SetPenState(pstate); {restore}
      DrawPicture(ph, r);

data 'PICT' (2)  0036 {size}  000A 0014 00AF 0078 {picFrame}
      1101 {version 1}  01 000A 0000 0000 00FA 0190 {clipRgn - 10 byte region}
      61 000A 0014 00AF 0078 0003 002D {paintArc rectangle,startangle,endangle}
      08 000A {pnMode patXor - note that the pnMode comes before the pnPat}
      09 AA55 AA55 AA55 AA55 {pnPat gray}
      69 0003 002D {paintSameArc startangle,endangle}
      FF {fin}
```



```
III. {CopyBits nopack, norgn, nowoman, nocry}
     GetPenState(pstate);
     SetRect(r, 20, 10, 120, 175);
     SetRect(smallr, 20, 10, 25, 15);
     SetRect(orgr, 0, 0, 30, 20);
     ClipRect(myWindow^.portRect);
     ph := OpenPicture(r);
     PaintRect(r);
     CopyBits (myWindow^.portBits, myWindow^.portBits,
               smallr, orgr, notSrcXor, NIL);
     {note: result BitMap is 8 bits wide instead of the 5 specified by smallr}
     ClosePicture;
     SetPenState(pstate); {restore the port's original pen state}
     DrawPicture(ph, r);

data 'PICT' (3)  0048 {size}  000A 0014 00AF 0078 {picFrame}
     1101 {version 1}  01 000A 0000 0000 00FA 0190 {clipRgn - 10 byte region}
     31 000A 0014 00AF 0078 {paintRect rectangle}
     90 0002 000A 0014 000F 001C {BitsRect rowbytes bounds (note that bounds is
                                  wider than smallr)}
     000A 0014 000F 0019 {srcRect}
     0000 0000 0014 001E {dstRect}
     00 06 {mode=notSrcXor}
     0000 0000 0000 0000 0000 {5 rows of empty bitmap (we copied from a
                               still-blank window)}
     FF {fin}
```

#22: TEScroll Bug

See also: TextEdit
Technical Note #131—TextEdit Bugs

Written by: Bryan Stearns
Updated:

April 21, 1986
March 1, 1988

A bug in TextEdit causes the following problem: a call to `TEScroll` with no horizontal or vertical displacement (that is, both `dh` and `dv` set to zero) results in disappearance of the insertion point. Since such calls do nothing, they should be avoided:

```
IF (dh <> 0) OR (dv <> 0) THEN TEScroll(dh,dv,myTEHandle);
```

#23: Life With Font/DA Mover—Desk Accessories

See also: The Resource Manager
Technical Note #6—Shortcut for Owned Resources

Written by: Ginger Jernigan
Updated:

April 25, 1985
March 1, 1988

This technical note describes how to make sure that your desk accessory will work after being moved by Font/Desk Accessory Mover.

If you want your desk accessory to work properly after being moved by the Font/DA Mover, there are some eccentricities that you need to be aware of. When the Font/DA Mover moves a desk accessory, it renumbers to avoid conflicts in ID numbers. It will also renumber all of your desk accessory's owned resources. See the Resource Manager chapter of *Inside Macintosh* for more information on owned resources.

Since these owned resources are renumbered, your code will need to calculate the resource ID of any owned resource it uses. For example, if your desk accessory has an owned 'DLOG' resource, and calls `GetNewDialog` with the ID you assigned to it originally, the Resource Manager will not find it. The solution is that every time your desk accessory references an owned resource, it must figure out (at execution time) the ID of the resource according to the current driver resource ID.

When the Font/DA Mover renumbers, it does its best to keep resources pointing to each other properly. This means that it tries to renumber resource IDs embedded in other resources as well as the resources themselves. For example, the reference to a 'DITL' within a 'DLOG' or 'ALRT' resource gets changed automatically. Font/DA Mover knows about the standard embedded resource IDs in most of the standard resources, but if you define your own, the Font/DA Mover won't be able to renumber them for you. The embedded resource IDs which the Font/DA Mover knows about are listed below.

Note that certain resources can never be owned, because their resource IDs are restricted to a certain range. One such example is a WDEF. Since the ID of a WDEF is specified along with a four bit variation code, the range of WDEF IDs that can be used is 0-16363. Since none of this falls within the owned resource ID range, WDEFs cannot be owned. For the same reason, MDEFs, CDEFs, and MBDFs can't be owned either.

As a rule of thumb, before you ship a desk accessory, move it to a disk with another desk accessory of the same ID. This will cause the Font/DA Mover to renumber your desk accessory. If the moved copy doesn't work, then there is probably something wrong with the way you are handling your owned resources.

Embedded resources known by Font/DA Mover

These are all true for Font/DA Mover 3.3 and newer:

- references to 'DITL' resources in 'DLOG'/'ALRT' resources
- references to 'ICON', 'PICT', 'CTRL' in 'DITL' resources
- references to 'MENU' resources inside the resources themselves (menuID field)
- references to 'MENU' resources in 'MBAR' resources

Anything not on this list has to be fixed by the desk accessory.

By the way...

Before Font/DA Mover, desk accessories could have an ID in the range 12 to 31. Now, and in the future, desk accessories can only have IDs in the range 12 to 26, because Font/DA Mover will only assign numbers in this range. Numbers 27 thru 31 are reserved.

#24: Available Volumes

See also: The File Manager

Written by: Bryan Stearns April 26, 1985
Modified by: Bryan Stearns October 15, 1985
Updated: March 1, 1988

Standard File lets the user select one file from any available volume; it is sometimes necessary for an application to find which volumes are present. This technical note gives the proper method of accomplishing this.

There is a little-noticed feature of the low-level file manager call `PBHGetVInfo` which allows specification of a "volume index" to select the volume. This volume index selects the nth volume in the VCB queue. The following function uses `PBHGetVInfo` to find out about a given volume. In MPW Pascal:

```
FUNCTION GetIndVolume(whichVol: INTEGER; VAR volName: Str255;
                      VAR volRefNum: INTEGER): OSErr;

{Return the name and vRefNum of volume specified by whichVol.}

   VAR
      volPB : HParamBlockRec;
      error : OSErr;

   BEGIN
      WITH volPB DO BEGIN          {makes it easier to fill in!}
         ioNamePtr  := @volName; {make sure it returns the name}
         ioVRefNum  := 0;         {0 means use ioVolIndex}
         ioVolIndex := whichVol; {use this to determine the volume}
      END; {with}
      error := PBHGetVInfo(@volPB,false); {do it}
      IF error = noErr THEN BEGIN      {if no error occurred }
         volRefNum := volPB.ioVRefNum; {return the volume reference}
      END; {if no error}
      {other information is available from this record; see the FILE}
      {Manager's description of PBHGetVInfo for more details...}
      GetIndVolume := error; {return error code}
   END;
```

In MPW C:

```
OSErr GetIndVolume(whichVol,volName,volRefNum)
short int whichVol;
char  *volName;
short int *volRefNum;

{
      /*Return the name and vRefNum of volume specified by whichVol.*/

      HVolumeParam      volPB;
      OSErr             error;


      volPB.ioNamePtr = volName; /*make sure it returns the name*/
      volPB.ioVRefNum = 0;       /*0 means use ioVolIndex*/
      volPB.ioVolIndex = whichVol; /*use this to determine the volume*/

      error = PBHGetVInfo(&volPB,false); /*do it*/
      if (error == noErr)    /*if no error occurred */
            *volRefNum = volPB.ioVRefNum; /*return the volume reference*/

      /*other information is available from this record; see the FILE*/
      /*Manager's description of PBHGetVInfo for more details...*/

      return(error);    /*always return error code*/
} /* GetIndVolume */
```

To find out about all volumes on-line, you can call this routine several times, starting at `whichVol` := 1 and incrementing `whichVol` until the routine returns `nsvErr`.

#25: Don't Depend on Register A5 Within Trap Patches

See also: The Operating System Utilities

Written by: Bryan Stearns June 25, 1986
Updated: March 1, 1988

Future software may allow desk accessories to have their own globals by changing register `A5` when the accessory is entered and exited. This can cause problems for applications that patch traps without following certain rules.

If your application patches any traps, it's important that the patches not depend on register `A5`. This is because you may have intercepted a trap used by a desk accessory.

If you need access to your globals within your patch, you can save `A5` (on the stack, perhaps), load `A5` from the low-memory global `CurrentA5` (this is guaranteed to be correct for your application), do whatever you have to do within your patch, then restore `A5` on the way out. Note that if you make any traps within your patch (or call the "real" version of the routine you patched), you should restore the caller's `A5` before doing so.

There are several ways of depending on `A5` within a patch that you should watch out for:

- Are you making any references to your global variables, or those of any units that you're using, such as `thePort` from QuickDraw? These are accessed with `A5`-relative references with negative offsets.
- Are you making any inter-segment subroutine calls? These are accessed with `A5`-relative references with positive offsets.
- Are you using any system calls (either traps or "glue" routines) which will depend on `A5` during their execution? In this case, you need to be sure that you restore the caller's `A5` before executing the call.

To be safest, patched traps should follow the same rules as interrupt handlers.

Note

In general, applications should not have to patch any traps, and risk compatibility problems if they do! If you'd like help in removing your dependence on patching, please contact Macintosh Developer Technical Support.

Macintosh
Technical Notes



Developer Technical Support

#26: Fond of FONDs

Written by: Joseph Maurer May 1992

This Technical Note takes the place of Tech Note #26, "Character vs. String Operations in QuickDraw" by Bryan Stearns (March 1988), which pointed out the possible differences between the results of a `StringWidth` call and successive calls to `CharWidth`. This Note updates and brings into a broader context the issues related to text measuring. It also provides additional documentation on font family resources (`'FOND'`s), and addresses various other frequently asked questions related to the Font Manager. For reasons of consistency and easier reference, much of the contents of Technical Notes #191, "Font Names," #198, "Font/DA Mover, Styled Fonts, and `'NFNT'`s," and #245, "Font Family Numbers," have been updated and worked into this Note as well.

Introduction

Every Macintosh developer needs to draw text in a GrafPort, and to specify typeface, size, and style. In most cases, there are no problems, and application developers don't need to have in-depth knowledge of the Font Manager's inner workings and the data structures involved. Sometimes, however, the results on the screen or on printed output may be different from what you expected. Then, usually, DTS comes into play to figure out what the problem is and how to fix it. This Note is based on sharp developer questions from the last year or so, which point mainly at shortcomings of the existing Font Manager architecture, inconsistencies in its data structures, and missing details in the documentation.

We'll start with a historical overview, which discusses the introduction of font family description resources (`'FOND'`s) back in 1986, explains the consequences of non-proportionally scaling fonts, and covers non-registration and volatility of font family numbers.

We will then deal with the Font/DA Mover and the built-in "Mover" of the Finder in System 7. We discuss a number of not-so-well-known aspects of moving fonts in and out of a suitcase file, and recommend that you altogether abandon the resource type `'FONT'`. We'll also comment on font names, and show you how to put separate stylistic variants of a typeface together into one font family. And we provide documentation on the `ffVersion` field of a `'FOND'` (accompanied by a disclaimer and another piece of irritating information).

The main body of this Note addresses how the Font Manager works in the `FMSwapFont` context, and gives information on the scaling factors in the `FMOutput` structure and on the changes introduced by TrueType. We again took the examples of unexpected behavior (under certain circumstances) from developer questions. Thanks for helping document this!

Determining the width of text, as required for line layout, is sometimes trickier than you might think. We will document the effects of `SetFractEnable` in more detail and mention some more line layout problems.

Finally, this Note includes sample code that puts the `OutlineMetrics` call to work, and determines text bounding boxes for bitmap fonts.

Some FOND Background

Originally (*Inside Macintosh* Volume I, Chapter 7), all font-related data was contained in resources of type `'FONT'`. For a font number within the range 0....255, and a font size restricted to less than 128, the (unnamed) `'FONT'` resource with an ID:

128*(font number) + (font size)

contained the bitmap font strike, while the `'FONT'` resource with ID = 128*(font number), corresponding to font size 0, did not contain any data, but its resource name provided the font family name. QuickDraw took care of stylistic variants like italic, bold, shadow, and so on; if a user had a specifically fine-tuned font strike for a stylistic variant, QuickDraw would **not** automatically substitute it when drawing text.

For aesthetic reasons, bitmap fonts for different sizes were usually designed with widths non-proportional to the point size. For example, the text *"Show the difference in text widths"* drawn with Courier 9 measures 170 pixels, whereas the same text drawn with Courier 18 measures 374 pixels, which is 10% more than you expect. (By the way, this is bad news for the ImageWriter printer driver. When "Best" mode (144 dpi) is selected and text in Courier 9 is to be printed, the printer driver uses Courier 18 to render the 9-point font size on the paper at twice the screen resolution, and obviously has big trouble compensating for the 10% difference in text width.)

On the other hand, given that only integer character widths (in QuickDraw's 72 dpi units) are possible, proportional font scaling is compromised anyway. Accumulated rounding errors in text measuring, particularly for scaled fonts, contribute to the headaches of many Macintosh programmers. The computed text widths (vital for positioning text precisely and for line layout algorithms to justify text) sometimes change quite abruptly when the user removes or adds certain font sizes.

The introduction of the LaserWriter, and the success of Macintosh in the desktop publishing arena, required an extension of the original Font Manager architecture. This extension is based on the concept of "font family description" resources of type `'FOND'`, and on a new resource type `'NFNT'` for the data of the existing `'FONT'` resources (see *Inside Macintosh* Volume IV, Chapter 5).

The `'FOND'` resource stores size-independent information about the font family, and its resource ID is the font number (in the range 0...32767). The resource name of the `'FOND'` is the font name, and it contains a variable-length **font association table**, which references the font strikes belonging to a specific font family. These references include size, style, and resource ID of the `'NFNT'` or `'FONT'` resource containing the bitmap font data. TrueType fonts were retrofitted into this scheme, and are identified as font strike resources for point size zero. Any reference to point size zero refers to a resource of type `'sfnt'`.

Note: The range 0...32767 for font numbers is subdivided into ranges for the various script systems (see *Inside Macintosh* Volume VI, pages 13-8 and 14-22, and Technical Note #242, "Fonts and the Script Manager"). This restricts the range of font numbers for the Roman script to 0...16383, with 0, 1, and 16383 reserved for the system.

Since Apple originally intended fonts to be referenced by their font family numbers, DTS attempted to register those numbers (see *Inside Macintosh* Volume I, page 219 and Volume IV, page 31). This failed—not only because the number of fonts registered grew greater than the number of font family numbers available, but also because the Font/DA Mover (version 3.8, shipped with System 6), and the "Mover" built into the System 7 Finder resolve conflicts between font IDs (which happened anyway!) by renumbering the fonts on-the-fly. There is no font ID registration any more—except for the very special case of Japanese Kanji `'FOND'`–`'fbit'` IDs, and potentially for Korean, Chinese and other double-byte fonts.

As early as April 1988, Technical Note #191, "Font Names," recommended the use of font names rather than font family numbers. Since then, the recommendation has been reinforced in *Inside Macintosh* Volume VI, page 12-16. Fortunately, most applications have been good about following this recommendation. Unfortunately, some exceptions remain, even in Apple's own software. QuickDraw Pictures created without 32-Bit QuickDraw refer to fonts by font family number only!

For obvious reasons of upward compatibility (to maintain existing fonts, and to avoid reflowing of existing documents), the introduction of `'FOND'`s did not solve all the problems. This is what this Note is all about.

Moofing Fonts

The Font/DA Mover utility has evolved into version 4.1, which knows about `'sfnt'`s. It is available on the *Developer CD Series* disc, path "Tools & Apps (Moof!): Misc. Utilities:". The Finder in System 7 incorporates its own "Mover" (see *Inside Macintosh* Volume VI, page 9-33), which makes the Font/DA Mover redundant for System 7 users.

Given the combinatorial explosion of all imaginable situations with `'FOND'`s, `'FONT'`s, `'NFNT'`s and `'sfnt'`s, and stylistic variations of fonts belonging to the same family, the font moving job deserves respect. The following notes cover some less well-known aspects of this business.

- If an old "standalone" `'FONT'` (without corresponding `'FOND'` resource) is moved into a suitcase file, Font/DA Mover or the System 7 Mover creates a minimal `'FOND'` resource on-the-fly. This `'FOND'` has no tables, and nearly all its fields are zeroed. The System 7 Finder also converts the resource type from `'FONT'` to `'NFNT'`; unfortunately, the Font/DA Mover keeps the resource type `'FONT'`.

  Note: While it is perfectly legal to have `'FOND'`s continue to reference the older `'FONT'` type, DTS recommends that you avoid `'FONT'`s. Accessing `'FONT'`s is much slower, since the Font Manager always looks for `'FOND'`s and `'NFNT'`s first. More importantly, `'FONT'`s are troublemakers if an application comes with its own font in its resource fork. Imagine an application that includes a private `'FOND'` which references a `'FONT'` in its resource fork by resource ID. When the Font Manager wants to load the font resource, it first looks for a resource of type `'NFNT'` with this same resource ID. If there's an `'NFNT'` in the System file with the same resource ID, the Font Manager will pick it instead of the `'FONT'` from the application's resource fork. This happens more often than you'd like to think!

- Under the current font architecture, the font name is the resource name of the `'FOND'` resource (let's forget about `'FONT'`s altogether), so the font name can be any Pascal string. Unfortunately, this conflicts with the 31-character limitation of a file name when the System 7 Finder derives the file name of a movable font file (*Inside Macintosh* Volume VI, page 9-34) from the font name. Some third-party fonts come with font names long enough to cause trouble. You may also see this problem when trying to open a suitcase if the Finder can't generate distinct names for all of the fonts in the suitcase; the Finder may say the suitcase is "damaged" when it is not.

  Note: Each TrueType `'sfnt'` resource contains a Naming Table (see *The TrueType™ Font Format Specification*, APDA™ M0825LL/A) which provides nearly unrestricted font naming capabilities, to accommodate the needs of font manufacturers. A forthcoming Macintosh Technical Note on TrueType Naming Tables gives additional information.

- QuickDraw and the current Font Manager have no provision for stylistic variants like "light," "medium," "demi," "book," "black," "heavy," "extra," "ultra," etc., used in the context of professional typesetting. Therefore, each of these variants comes with a separate font family resource. Probably for reasons of consistency, the "italic" variants have their own font family resources as well. Unfortunately, unless each `'FOND'` references both the "plain" and the "italic" font strikes, QuickDraw will no longer know a customized italic font strike exists.

  It is fairly easy, using System 7 and ResEdit, to merge two font families (named, for exmaple, "myFont" and "myFont italic") into one. This way, QuickDraw will automatically use the pre-designed italic font strike instead of creating one algorithmically. Follow these convenient steps:

  1. Make sure there is no resource ID conflict between the `'NFNT'`s and `'sfnt'`s belonging to both families.
  2. Make sure the style bits for italic are set in the font association table of "myFont italic."
  3. From ResEdit's File menu, "Get Info..." on the "myFont" `'FOND'` resource. Write down the resource ID of the "myFont" `'FOND'`.
  4. From ResEdit's File menu, "Get Info..." on the "myFont italic" `'FOND'`. Change its resource ID to be identical to the one you wrote down in step 3. Change its resource name to "myFont."
  5. Use the Finder in System 7 to move the contents of the "myFont italic" suitcase into the original "myFont" suitcase. It will merge all constituents into one font association table, and thus enable transparent substitution of the right font for QuickDraw's italic style.

Version Numbers

The `'FOND'` structure (see *Inside Macintosh* Volume IV, page 45, "FamRec") contains a field `ffVersion`, and inquiring minds naturally want to know more about it. Before anything else, however, please read the following disclaimer:

Disclaimer: The Font Manager does not check version numbers in a `'FOND'`, and we recommend that you not rely on the (intentionally vague) statements below, but rather analyze the data in the `'FOND'` independently.

Currently, values 0...3 may appear in the `ffVersion` field, with the following intended interpretations:

Version 0: Usually indicates that the `'FOND'` has been created on the fly by the Font/DA Mover (or the System 7 Finder). But the `'FOND'` for Palatino on the distribution disks of System 7 is a counterexample.

Version 1: Obviously indicates the first version when `'FOND'`s came out (*Inside Macintosh* Volume IV, page 36).

Version 2: Corresponds to the extension of the `'FOND'` format documented in *Inside Macintosh* Volume V, page 185 (which does not mean that the `'FOND'` actually contains a bounding box table).

Version 3: The `'FOND'` is supposed to contain a bounding box table.

This brings up an annoying fact. All measurement values (referring to a hypothetical 1-point font) in the `'FOND'` are in a 16-bit fixed-point format, with an integer part in the high-order 4 bits and a fractional part in the low-order 12 bits. You would expect that negative values (like for `ffDescent`, or in the kerning tables) are represented in the usual two's-complement format, such that standard binary arithmetic applies. This is mostly true, but not always. Again, `Palatino` is a counterexample (and probably not the only one). To our knowledge, version 0 and version 1 `'FOND'`s have negative values represented in a format where the most significant bit is the sign bit, and the rest represents the absolute value. However, there is nothing in the system software that enforces this, so counterexamples may exist.

Warning: Don't rely on the version number, but include sanity checks for the negative values in a `'FOND'` instead! The following Pascal function shows how this can be done:

```
FUNCTION Check4p12Value(n: Integer): Integer;
{ n is a 4.12 fixed-point value; i.e., its "real" value is n/4096.     }
{ If n is "unreasonably negative," interpret the most significant bit }
{ as sign bit, and convert to the usual two's complement format.       }

   BEGIN
      IF n < $8FFF THEN { means: (4.12-interpretation of n) is below - 7 }
         Check4p12Value := - BitAnd(n,$7FFF)
       { i.e., mask sign bit, and take negative of absolute value }
       ELSE
         Check4p12Value := n;
   END;
```

In the Heart of the Font Manager

Swapping Fonts

As stated in *Inside Macintosh,* there is only one contact between QuickDraw and the Font Manager: the `FMSwapFont` function. Each of the three QuickDraw text *measuring* functions (`CharWidth`, `StringWidth` and `TextWidth`) always ends up in the QuickDraw bottleneck procedure `QDProcs.txMeasProc`. Each of the three QuickDraw text *drawing* procedures (`DrawChar`, `DrawString` and `DrawText`) always ends up in the `QDProcs.textProc` bottleneck procedure. Any reasonable `textProc` (like `StdText`) needs to call the currently-installed text measuring bottleneck procedure before actually rendering the text. And what does any reasonable text measuring bottleneck procedure (like `StdTxMeas`) do first, before anything

else? It calls FMSwapFont, to make sure we are talking about the right font and its properties! (To be precise, GetFontInfo and FontMetrics are the other calls that make sure the right font is swapped in and set up, without requiring you to call FMSwapFont explicitly.)

Responding to a font request is a lot of work, and FMSwapFont has been optimized to return as quickly as possible if the request is the same as the previous one. Building the **global width table** (see *Inside Macintosh* Volume IV, page 41) is among the more time-consuming tasks related to FMSwapFont; this is why the Font Manager maintains a cache of up to 12 width tables.

Inside Macintosh Volume I, page 220 documents the Font Manager's choice when a font of the requested size is not available. However, some consequences or additional features have occasionally been a surprise to developers (and users as well).

Scaling Factors in FMOutPut and StdTxMeas

Let's suppose you have only a 12-point bitmap version of Palatino, and don't have any Palatino outline fonts. When you request Palatino 18, QuickDraw sets up the FMInput record with size = 18 and numer = denom = Point($00010001). On return, the FMOutput record contains the handle to the font record to use (the 'NFNT' with the Palatino 12 bitmap font strike), and indicates the scaling factors QuickDraw will have to use to produce the desired text point size in FMOutput.numer and FMOutput.denom. In this example, that ratio is 3/2.

Note that these are also the values returned in StdTxMeas (*Inside Macintosh* Volume I, page 199) if you call the procedure with numer = denom = Point($00010001). Why? Because StdTxMeas calls FMSwapFont, as explained under "Swapping Fonts." StdTxMeas does **not** apply these scaling factors to the text it measures. In our example, it would measure Palatino 12 and return numer and denom in the ratio 3/2 to tell you that your application must multiply the results by these values to get the correct measurements for Palatino 18. This has surprised more than one programmer who didn't expect numer and denom to change!

By the way, the Font Manager always normalizes the scaling factors as fractions numer/denom such that the denominator is equal to 256. In our example, the real numbers returned by FMSwapFont or StdTxMeas are numer = 384 and denom = 256.

Warning: If the scaling factors numer and denom passed to StdTxMeas, StdText (see *Inside Macintosh* Volume I, pages 198 and 199), or in the FMInput record to FMSwapFont are such that txSize*numer.v/denom.v is less than 0.5 and rounds to 0, and if there is more than one 'sfnt' resource referenced in the font association table, then the current Font Manager may get confused and return results for the wrong font strike.

TrueType Always Has the Right Size

The default value of outlinePreferred is FALSE. If you have bitmap fonts for Palatino 12 and Palatino 14 in your system as well as a Palatino TrueType font, then requests for Palatino 12 or Palatino 14 are fulfilled with the bitmap fonts, but requests for any other size are fulfilled with the TrueType font. In particular, if you (or, for example, a printer driver) need Palatino 12 scaled by 2, the Font Manager will actually look for Palatino 24 and return the outline font, regardless of the setting of outlinePreferred. Even if you wanted the bitmap font doubled for exact

"what-you-see-is-what-you-get" text placement, you're out of luck—you get the TrueType font, which may have very different font metrics or character shapes.

If the Font Manager uses an outline font to fulfill a given font request, the `IsOutline` function returns `TRUE`. Interestingly, this does not imply that `RealFont` returns `TRUE` as well. If the text size is smaller than the value `lowestRecPPEM` ("smallest readable size in pixels") in the `'head'` font header in the TrueType font (see *The TrueType Font Format Specification*, version 1.0, page 227), then `RealFont` returns `FALSE`!

First Size, Then Style—or: To Be or Not to Be Outline

When the Font Manager walks the font association table of a `'FOND'` to look for a font strike of a specified size and style, it stops at the first font of the right size. Only if you requested a stylistic variant (like bold or italic) does it take a closer look at the fonts of the same size. It does this by putting weights on the various style bits (for example, 8 for italic, 4 for bold, 3 for outline) and choosing the font strike whose style weight most closely matches the weight of the requested style. All this is fine when only bitmap fonts are available. With the presence of TrueType outlines, however, the results are not always as expected, depending on the font configuration installed.

Let's look at a few examples:

Example 1: Let's suppose you have the bitmap font Times 12 (Normal) and the TrueType fonts Times (Normal), Times Italic and Times Bold in your system. If you request Times 14 Italic or Times 14 Bold, it's rendered from the Times Italic or Times Bold TrueType fonts. However, if you ask for Times 12 Italic or Times 12 Bold, and your system has the default setting of `outlinePreferred = FALSE`, the Font Manager decides to take the Times 12 bitmap and let QuickDraw algorithmically slant it (for italics) or smear it (for bold).

Example 2: Let's suppose you want to draw big, bold Helvetica characters and there are no existing bitmaps for the size you want. If the Helvetica Bold TrueType outlines are available, the Font Manager chooses them and the only surprise in text rendering will be a pleasant one. If there is no Helvetica Bold TrueType font, however (like in the machine of your customer, who kept only the normal Helvetica TrueType font in his system), then the characters are rendered using the normal Helvetica outlines and, in a second step, QuickDraw applies its horizontal 1-pixel "smearing" to simulate the bold stylistic variant. The result is very different (and rather an unpleasant surprise).

Example 3: Admittedly, this is less likely (but it has happened). Let's suppose somebody decides to rip the Times TrueType outline out of the System file (don't ask me why—I don't know). He forgets to take the Times Italic TrueType outline away as well. The next time he draws text in Times (Normal), in a size for which there is no bitmap font (or if `outlinePreferred = TRUE`), the Font Manager goes for an `'sfnt'`, and the text shows up in *italic* (what a surprise!).

Unfortunately, given the current implementation of the Font Manager, there are no solutions to the problems illustrated above—other than asking users of your application to install the fonts you recommend. The only way to anticipate these potential surprises from within your application is to

look into the 'FOND's font association table. You can't depend on the IsOutline function because it returns TRUE as soon as the Font Manager stops at an 'sfnt', in its first pass through the font association table—regardless of subsequent stylistic variations. This means, for example, if you ask for Helvetica Bold and IsOutline returns TRUE, you don't know if you got the Helvetica Bold TrueType font or if QuickDraw "smeared" the Helvetica (Plain) TrueType font.

Where Do the Widths Come From?

Text measuring (for example, for precise text placement in forms with bounding boxes) and most line layout algorithms for justified text rely heavily on the character widths contained in the global width table. Given that under the current font architecture, we may easily have three or more different width tables for the same font specification (the non-proportional integer widths attached to the 'NFNT', the fractional widths contained in the 'FOND', and the fractional widths provided by the 'sfnt'), it is important to understand where the widths come from in any case.

Since SetFractEnable was introduced (*Inside Macintosh* Volume IV, page 32 and Volume V, page 180), its setting TRUE or FALSE was supposed to give predictable effects. If it's FALSE, the Font Manager takes the integer widths from the 'NFNT'; if it's TRUE, it takes the fractional widths from the 'FOND'. Unfortunately, there are some additional details and side effects that are not well known.

- The Font Manager looks at bit 14 of the ffFlags field in the 'FOND' (see *Inside Macintosh* Volume IV pages 36 and 37). If it is set (like it is for Courier), the fractional widths from the 'FOND' are *never* used.
- If SetFractEnable is TRUE and you request a stylistic variation like bold or italic, the Font Manager looks at bits 12 and 13 of the ffFlags field to decide how different widths or extra widths for the stylistic variants have to be used. What it decides is documented in the "Font Manager" chapter of *Inside Macintosh Preview*, located on the *Developer CD Series* discs.
- Given that it is not possible to set the pen to a fractional position, precise text positioning with fractional widths enabled is always compromised because of (accumulated) rounding errors.
- QuickDraw distributes the accumulated rounding errors across characters within a string (instead of adding it at the end of the drawn text). This results in poor text quality on the screen, and in problems when calculating the position of the insertion point between characters.
- The LaserWriter driver watches what you pass to SetFractEnable. Passing TRUE to SetFractEnable disables some of the LaserWriter driver's line layout features, assuming that the programmer intends to control text placement manually. Explicitly passing FALSE to SetFractEnable achieves different results than using the default value of FALSE—Font Substitution behaves differently, for example. These effects are sometimes Not What You Wanted.
- On non-32-Bit-QuickDraw systems, SetFractEnable is not recorded in pictures. This affects the line layout of text reproduced through DrawPicture if the picture was created with fractional widths enabled.

In systems with TrueType, quite naturally the widths *always* come from the 'sfnt' when the Font Manager uses a TrueType font. If fractEnable is FALSE, hand-tuned integer character widths for specific point sizes come from the 'hdmx' table in the 'sfnt'. If fractEnable is FALSE and no 'hdmx' table is present or it contains no entries for the desired point size, the fractional character widths from the 'sfnt' are rounded to integral values.

More Line Layout Problems

The routines `SpaceExtra` (*Inside Macintosh* Volume I, page 172) and `CharExtra` (*Inside Macintosh* Volume V, page 77; available only in color GrafPorts) are intended to help you draw fully justified text. This works fine on the screen, but not all printer drivers are smart enough to use these settings appropriately under all circumstances. In particular, if you pass `TRUE` to `SetFractEnable`, or if you turn the LaserWriter driver's line layout algorithm off (by means of the picture comment `LineLayoutOff`; see Macintosh Technical Note #91), or if font substitution is enabled and actually occurs, it is better not to rely on `SpaceExtra` and `CharExtra` when printing fully justified text. Instead, keep the LaserWriter driver's line layout adjustments off, and calculate the placement of your text (word by word, or even character by character) yourself.

Putting Text Into Boxes

TrueType fonts came to the Macintosh together with seven new Font Manager routines (as documented in *Inside Macintosh* Volume VI, Chapter 12). The `OutlineMetrics` function is certainly the most sophisticated of these, and sample code illustrating its usage may be helpful. The following procedure `DrawBoxedString` assumes that the new outline calls (*Inside Macintosh* Volume VI, Chapter 12) are available, and that `IsOutline` returns `TRUE` for the current port setting.

```
PROCEDURE DrawBoxedString(pt: Point; s: Str255);
{ Draw string s at pen position (pt.h, pt.v), and show each character's bounding box. }

   CONST
      kOneOne = $00010001;

   VAR
      advA: FixedPtr;
      lsbA: FixedPtr;
      bdsA: RectPtr;
      err,i,yMin,yMax,leftEdge,temp: Integer;
      numer,denom: Point;
      advance,lsb: Fixed;
      r: Rect;

   BEGIN
      numer := Point(kOneOne);
      denom := Point(kOneOne); { unless you want to draw with scaling factors
                                 .... }
      MoveTo(pt.h,pt.v);
      DrawString(s);
{ This is for the pleasure of your eyes only - in practice, you would probably }
{ first look at the metrics, and then decide where and how to draw the string! }
      advA := FixedPtr(NewPtr(Length(s) * SizeOf(Fixed)));
      lsbA := FixedPtr(NewPtr(Length(s) * SizeOf(Fixed)));
      bdsA := RectPtr(NewPtr(Length(s) * SizeOf(Rect)));
      { Please, check for NIL pointers here! }
      err := OutlineMetrics(Length(s),@s[1],numer,denom,yMax,yMin,advA,lsbA,
                            bdsA);
      advance := 0;
      FOR i := 1 TO Length(s) DO { for each character }
         BEGIN
         { Add accumulated advanceWidth and leftSideBearing of current glyph }
         { horizontally to starting point. }
         leftEdge := pt.h + Fix2Long(advance + lsbA^);
```

```
            r := bdsA^; { The bounding box rectangle is in TrueType coordinates. }
            temp := r.bottom; { need to flip it "upside down" }
            r.bottom := - r.top;
            r.top := - temp;
            OffsetRect(r,leftEdge,pt.v);
            FrameRect(r); { This is the glyph's bounding box. }
            advance := advance + advA^;
            { "Advance" is Fixed, to avoid accumulation of rounding errors. }
            { Now, bump pointers for next glyph. }
            bdsA := RectPtr(ord4(bdsA) + SizeOf(Rect));
            advA := FixedPtr(ord4(advA) + SizeOf(Fixed));
            lsbA := FixedPtr(ord4(lsbA) + SizeOf(Fixed));
         END;
         DisposPtr(Ptr(advA));
         DisposPtr(Ptr(lsbA));
         DisposPtr(Ptr(bdsA));
      END; { DrawBoxedString }
```

`OutlineMetrics` exists because many developers need pixel-precise information on placement and bounding boxes, often on a character-by-character basis. Unfortunately, there is no similar facility for text drawing with bitmap fonts. Worse, under certain circumstances, italicized or shadowed (or both) bitmap fonts are sometimes poorly clipped, particularly for scaled sizes. Cosmetic workarounds include adding a space character to strings drawn in italic. You might also draw the text off-screen first (in order to determine the bounding box of the black pixels) and use `CopyBits` to copy the text onto the screen—but using `CopyBits` for text is usually bad for printing.

The existing documentation on the `FMOutput` and global width table structures (*Inside Macintosh* Volume I, page 227 and Volume IV, page 41) suggests it's possible to devise a routine for determining a fairly precise text bounding box for bitmap fonts. The procedure below, `BitmapTextBoundingBox`, is a first attempt. It assumes that TrueType is unavailable, or that the `IsOutline` call returned `FALSE` for the current port settings. While the returned bounding box is not always "tight," be careful before modifying the algorithm and shrinking the resulting bounding box—bitmap fonts just don't contain enough precise information for an exact bounding box, and different bitmap fonts and different sizes may require different adjustments.

```
PROCEDURE TextBoundingBox(s: Str255; numer,denom: Point; VAR box: Rect);

   CONST
      FMgrOutRec = $998; { FMOutRec starts here in low memory }
      tabFont = 1024;
      { global width table offset for font record handle, see IM IV-41 }

   TYPE
      FontRecPtr = ^FontRec;

   VAR
      hScale,vScale: Fixed;
      err,intWidth,kernAdjust: Integer;
      xy: Point;
      info: FontInfo;  { only for StdTxMeas; we'll use FontMetrics }
      fm: FMetricRec;  { see Inside Macintosh, IV-32 }
      fmOut: FMOutput;
      h: Handle;

   BEGIN
      intWidth := StdTxMeas(ord(s[0]),@s[1],numer,denom,info);
      { calls FMSwapFont and everything - }
      { StdTxMeas returns possibly modified scaling factors numer, denom }
      hScale := FixRatio(numer.h,denom.h);
```

```
    vScale := FixRatio(numer.v,denom.v);
    { These are the scaling factors QuickDraw uses    }
    { in "stretching" the available character bitmaps }
    fmOut := FMOutPtr(FMgrOutRec)^;
    { has been filled by the most recent FMSwapFont,  }
    { implicitly called by StdTxMeas  }
    SetRect(box,0, - info.ascent,intWidth,info.descent);
    { bounding box for unscaled plain text }
    IF (italic IN thePort^.txFace) AND (fmOut.italic <> 0) THEN BEGIN
    { the following is heuristics … }
       box.right := box.right + (info.ascent + info.descent - 1) *
                    fmOut.italic DIV 16;
       FontMetrics(fm);
       HLock(fm.WTabHandle); { We'll point to global WidthTable. }
       h := Handle(LongPtr(ord4(fm.WTabHandle^) + tabFont)^);
       { Be sure it's a handle to a 'NFNT' or 'FONT' ! }
       kernAdjust := FontRecPtr(h^)^.kernMax;
       OffsetRect(box, - kernAdjust,0);
       HUnlock(fm.WTabHandle);
    END;
    IF (bold IN thePort^.txFace) AND (fmOut.bold <> 0) THEN
       box.right := box.right + fmOut.bold - fmOut.extra;
    IF (outline IN thePort^.txFace) THEN InsetRect(box, - 1, - 1);
    IF (shadow IN thePort^.txFace) AND (fmOut.shadow <> 0) THEN BEGIN
       IF fmOut.shadow > 3 THEN fmOut.shadow := 3;
       box.right := box.right + fmOut.shadow;
       box.bottom := box.bottom + fmOut.shadow;
       InsetRect(box, - 1, - 1);
    END;
    { Now scale the box (more or less) as QuickDraw would do. }
    { Note that some of the adjustments are based on trial and error… }
    box.top := FixRound(FixMul(Long2Fix(box.top),vScale));
    box.left := FixRound(FixMul(Long2Fix(box.left),hScale)) - 1;
    box.bottom := FixRound(FixMul(Long2Fix(box.bottom),vScale)) + 1;
    box.right := FixRound(FixMul(Long2Fix(box.right),hScale)) + 1;
    GetPen(xy);
    OffsetRect(box,xy.h,xy.v);
END;
```

Conclusion

At the time when the original Font Manager architecture was designed, based on QuickDraw's hard-coded 72 dpi resolution, nobody could anticipate that some years later, the Macintosh would be used to tackle professional typesetting projects. Several advanced page layout applications managed to work around the "built-in" limitations, at high development costs, and some compatibility and performance problems. In many other cases, however, those limitations caused questions to DTS and unsatisfying compromises. This Note can't do much more than explain the state of affairs; the real solution to the problems must come from a redesigned foundation. TrueType leads the way and already fulfills many of the requirements; everything else is getting closer and closer.

Further Reference:

- *Inside Macintosh*, Volume I, Chapter 7, The Font Manager
- *Inside Macintosh*, Volume IV, Chapter 5, The Font Manager
- *Inside Macintosh*, Volume V, Chapter 9, The Font Manager
- *Inside Macintosh*, Volume VI, Chapter 12, The Font Manager
- *New & Improved Inside Macintosh*, Imaging: The Font Manager. *Developer CD Series* disc, path Developer Essentials: Technical Docs: Inside Macintosh Preview
- Macintosh Technical Note #91, Picture Comments—The Real Deal
- Macintosh Technical Note #191, Font Names
- Macintosh Technical Note #242, Fonts and the Script Manager
- Macintosh Technical Note #245, Font Family Numbers
- *Apple LaserWriter Reference*, Chapter 2, Working With Fonts (Addison-Wesley, 1988)
- Adobe Technical Note #0091 (PostScript Developer Support Group), Macintosh FOND Resources

PostScript and Adobe are registered trademarks of Adobe Systems Incorporated.
Helvetica and Palatino are registered trademarks of Linotype AG and/or its subsidiaries.

Velocio is **not** a trademark of the author.



#27: MacDraw's PICT File Format

Revised: August 1989
Written by: Ginger Jernigan August 1986

This Technical Note formerly described the PICT file format used by MacDraw® and the picture comments the MacDraw used to communicate with the LaserWriter driver.
Changes since March 1988: Updated the CLARIS address.

This Note formerly discussed the PICT file format used by MacDraw, which is now published by CLARIS. For information on MacDraw (its specific use of the PICT format) and other CLARIS products, contact CLARIS at:

CLARIS Corporation
5201 Patrick Henry Drive
P.O. Box 58168
Santa Clara, CA 95052-8168

Technical Support
Telephone: (408) 727-9054
AppleLink: Claris.Tech

Customer Relations
Telephone: (408) 727-8227
AppleLink: Claris.CR

Inside Macintosh, Volume V–39, Color QuickDraw and Technical Note #21, QuickDraw's Internal Picture Format, now document the PICT file format. Technical Note #91, Optimizing for the LaserWriter—Picture Comments, now documents the picture comments which the LaserWriter driver supports.

Further Reference:

- *Inside Macintosh*, Volume V–39, Color QuickDraw
- Technical Note #21, QuickDraw's Internal Picture Format
- Technical Note #91, Optimizing for the LaserWriter—Picture Comments

MacDraw is a registered trademark of CLARIS Corporation.

#28: Finders and Foreign Drives

Written by: Ginger Jernigan
Updated:

May 7, 1984
March 1, 1988

This technical note describes the differences in the way the 1.1g, 4.1, 5.0 and newer Finders communicate with foreign (non-Sony) disk drives.

Identifying Foreign Drives

Non-Sony disk drives can send an icon and a descriptive string to the Finder; this icon is used on the desktop to represent the drive. The string is displayed in the "Get Info" box for any object belonging to that disk. When the Finder notices a non-Sony drive in the VCB queue, it will issue 1 or 2 control calls to the disk driver to get the icon and string.

Finder 1.1g issues one control call to the driver with `csCode` = 20 and the driver returns the icon ID in `csParam`. This method has problems because the icon ID is tied to a particular system file. So, if the Finder switch-launches to a different floppy, the foreign disk's icon reverts to the Sony's.

Finders 4.1 and newer issue a newer control call and, if that fails, they issue the old Control call. The new call has `csCode` = 21, and the driver should return a pointer in `csParam`. The pointer points to an 'ICN#' followed by a 1 to 31 byte Pascal string containing the descriptor. This implies that the icon and the string must be part of the disk driver's code because only the existence of the driver indicates that the disk is attached.

This has implications about the translation of the driver for overseas markets, but the descriptor will usually be a trademarked name which isn't translated. However, the driver install program could be made responsible for inserting the translated name into the driver.

Drivers should respond to both control calls if compatibility with both Finders is desired.

Formatting Foreign Drives

When the user chooses the Erase Disk option in the Finder, a non-Sony driver needs to know that this has happened so it can format the disk. Finder 4.1 and newer notify the driver that the drive needs to be formatted and verified. They first issue a `Control` call to the driver with the `csCode` = 6 to tell the disk driver to format the drive. Then they issue a `Control` call with a `csCode` = 5 to tell the driver to verify the drive.

Other Nifty Things to Know About

Finders 4.1 and newer also permit the user to drag any online disk to the trash can. The Finder will clean up the disk state, issue an `Eject` call followed by an `Unmount` call to the disk and then, an event loop later, reclaim all the memory. This means any program/accessory used to mount volumes should reconcile its private data, menus, etc. to the current state of the VCB queue. These Finders also notice if a volume disappears and will clean up safely. But, because of a quirk in timing, a mount manager cannot unmount one volume then mount another immediately; it must wait for the Finder to loop around and clean up the first disk before it notices the second. (It should have cleaned up old ones before it notices new ones, but it doesn't.)

Finders 5.0 and newer allow you to drag the startup disk to the trash; Finder 4.1 just ignored you. Finders 5.0 and newer take the volume offline as if you had chosen Eject.

#29: Resources Contained in the Desktop File

See also: The Finder Interface

Written by: Ginger Jernigan May 7, 1985
Modified by: Ginger Jernigan December 2, 1985
Updated: March 1, 1988

This technical note describes the resources found in the Desktop file. **Note:** Don't base anything critical on the format of the Desktop file. AppleShare already uses another scheme; AppleShare volumes don't have Desktop files. The format of this file can, and probably will, change in the future.

The Desktop file contains almost the same resources for both the Macintosh File System (MFS) and the Hierarchical File System (HFS). This technical note describes the resources found in both. This information is for reading only. This means your application can read it but it should NEVER write out information of its own, because the Finder, as well as Macintosh Developer Technical Support, won't like it.

The Desktop is a resource file which contains the folder information on an MFS volume, the "Get Info" comments, the application bundles, 'FREF's and 'ICN#'s, and information concerning the whereabouts of applications on an HFS disk. Everything except the comments are preloaded when the desktop is opened, making it easier for the Finder to find things.

The contents of the Desktop file are described below. The resource types are the same for both MFS and HFS volumes unless otherwise stated.

'APPL': This resource type is used by the HFS to locate applications. This is used by the Finder to locate the right application when a document is opened. Each application is identified by the creator, the directory number, and the application name. This is used only by HFS.

'BNDL': This resource type contains a copy of all of the bundles for all of the applications that are either on the disk or are the creators of documents that are on the disk. This is used by the Finder to find the right icons for documents and applications. If you have a document whose creator the Finder has not seen yet, it will not be in the Desktop file and the default document icon will be used.

'FREF': This contains a copy of all of the FREFs referenced in the bundles.

'FCMT': This resource contains all of the "Get Info" comments for applications and documents. On MFS volumes the ID is a hash of the object's name. The hashing algorithm is as follows:

```
; FUNCTION HashString(str: Str255): INTEGER;

; The ID for the FCMT returned in function result

HashString
      MOVE.L    (SP)+,A0        ; get return address
      MOVE.L    (SP)+,A1        ; get string pointer

      MOVEQ     #0,D0           ; get string length
      MOVE.B    (A1)+,D0

      MOVEQ     #0,D2           ; accumulate ID here
@2
      MOVE.B    (A1)+,D1        ; get next char
      EOR.B     D1,D2           ; XOR in
      ROR.W     #1,D2           ; stir things up
      BMI.S     @1              ; ID must be negative
      NEG.W     D2
@1
      SUBQ.W    #1,D0           ; loop until done
      BNE.S     @2              ; until end of string

      MOVE      D2,(SP)         ; return the hashed code
      JMP       (A0)
```

For HFS volumes, the ID of the resource is randomly generated using `UniqueID`. To find the ID of the comment for a file or directory call `PBGetCatInfo`. The comment ID for a file is kept in `ioFlXFndrInfo.fdComment`. The comment ID for a directory is kept in `ioDrFndrInfo.frComment`.

'FOBJ': This resource type contains all of the folder information for an MFS volume. The format of this resource is not available. This is only in an MFS volume's Desktop file.

'ICN#': This resource type contains a copy of all of the 'ICN#' resources referenced in the bundles and any others that may be present.

'STR ': This is a string that identifies the version of the Finder, but it isn't always correct.

Creators: A resource with a type equal to the creator of each application with a bundle is stored in the Desktop file for reference purposes only. The data stored in these resources is for the Finder's use only.

Be aware that if a resource is copied from an application resource file and there is an ID conflict, the Finder will renumber the resource in the Desktop file.

#30: Font Height Tables

See Also: The Font Manager
The Resource Manager

Written by: Gene Pope
Updated:

April 25, 1986
March 1, 1988

This technical note describes how the Font Manager (except in 64K ROMs) calculates height tables for fonts and how you can force recalculation.

In order to expedite the processing of fonts, the Font Manager (in anything newer than the 64K ROMs) calculates a height table for all of the characters in a font when the font is first loaded into memory. This height table is then appended to the end of the font resource in memory; if some program (such as a font editor) subsequently saves the font, the height table will be saved with the font and will not have to be built again. This is fine for most cases except, for example, when the tables really should be recalculated, such as in a font editor when the ascent and/or descent have changed.

The following is an example of how to eliminate the height table from a font:

```
IF (BitAnd(hStrike^^.fontTyp,$1)=1) THEN BEGIN {We have a height table}
   {Truncate the height table}
   SetHandleSize(Handle(hStrike),GetHandleSize(Handle(hStrike)-
                 (2*(hStrike^^.lastChar-hStrike^^.firstChar)+3)));
   {We no longer have a height table so set the flag to indicate that}
   hStrike^^.format := BitAnd(hStrike^^.fontType,$FFFFFFFE);
END;
```

In MPW C:

```
if ((**hStrike).fontType & 0x1 ==1) { /*We have a height table*/
   /*Truncate the height table*/
   SetHandleSize((Handle)hStrike,GetHandleSize((Handle)hStrike)-
                 (2*((**hStrike).lastChar-(**hStrike).firstChar)+3));
   /*We no longer have a height table so set the flag to indicate that*/
   (**hStrike).fontType = (**hStrike).fontType & 0xFFFFFFFE;
}
```

where `hStrike` is a handle to the 'FONT' or 'NFNT' resource (handle to a `FontRec`).

Note: After the height table has been eliminated, the modified font should be saved to disk (with `ChangedResource` and `WriteResource`) and purged from memory (using `ReleaseResource`). This is an important step, because the Font Manager does not expect other code to go behind its back removing height tables that it has calculated.



#31A: GestaltWaitNextEvent

Revised by: C.K. Haun <TR> April 1 1992

This Technical Note discusses a new Event Manager call in Macintosh System Software.

The Changing World

The Macintosh operating environment is changing rapidly. Modular system software, dynamically linked libraries, plug and play hardware, all add up to a confusing environment for the application programmer.

To dispel this confusion, it is essential that an application *always* know what features are available for its use. The user experience will be greatly enhanced when the user can drop a new system extension into their System Folder and immediately use it in all applications.

Review

To allow this, a new function (provided as a system extension) has been added to System 7 and later, `GestaltWaitNextEvent`.

The best way to explain GWNE is to see it in action. The function prototype for `GWNE` is:

```
pascal EventReturnStructHandle GestaltWaitNextEvent(EventMaskHandle
theMask,SleepHandle sleepValue,GestaltAvailableHandle
featuresAvailable,GestaltAvailableHandle minimumNeeded,GWNECallbackHandle
myCallBack);
```

Draft

The first thing you'll notice is that the mouse region parameter is missing. No one could ever figure this out, so it's been dropped.

There are six new structures defined for this call.

The first is the `EventReturnStruct`. Since you never know what features may be connected to your Mac, you can never be certain what events you'll get back. Also, it is possible to get multiple events simultaneously, depending on the types of devices and extensions the user has installed So this variable structure has been created to let you know what happened during the event call.

```
struct EventReturnStruct{
      unsigned long           NumberOfEvents;
      struct EventRecord2     **theEvents;
};
```

where EventRecord2 is:

```
struct EventRecord2 {
      unsigned long     typeOfEvent;
      Handle            eventData;
      DateTimeRec       eventTime;
      EventRecord2      **nextEvent;
```

```
};
```

When GWNE returns, you will then walk through the linked list of EventRecord2 structures, examining the event type and parsing the data in the eventData field as appropriate for that event. The numberOfEvents parameter is available to quickly determine how many events have occurred. Since it is possible for you to get up to 4294967295 events per GWNE call (or up to available memory) it may be appropriate to display a watch cursor or 'please wait' dialog after returning from GWNE.

Also please note that each event contains a DateTimeRec structure. Ticks are not enough for some events, for example if the SubSpace manager (see *develop* issue 7) is installed, the normal starting point of Jan 1 1904 is not adequate, since events posted many millennia earlier or later may also be queued to your machine. Please see the specific event source documentation for explanation of this record for specific events.

The next new structure is the EventMaskStruct. This is necessary since there is a large amount of possible events (again, up to 4294967295) that you may be interested in, and they may have different masking needs.

```
struct EventMaskStruct{
      unsigned long           typeOfEvent;
      Handle                  eventAcceptParameters;
      Handle                  eventRefuseParameters;
      struct EventMaskStruct  **nextEventMask;
};
```

Review

You'll note that you can pass reasons both for accepting or refusing any event, the contents of these handles is determined by the eventType field.

Warning: You *must* pass a handle in both eventAcceptParameters and eventRefuseParameters. Failure to do so may cause an event not intended for your computer to be accepted.

Draft

The old sleep value has also changed. The new structure SleepHandle defines not only how long you'll sleep, but also if you should wake up for any specified event. This gives you much more flexibility to customize your application to meet the real needs of your customers.

```
struct SleepStruct{
      unsigned long           typeOfEvent;
      Handle                  eventWakeParameters;
      Handle                  eventStayAsleepParameters;
      EventMaskHandle         XOREvents;
      EventMaskHandle         ANDEvents;
      EventMaskHandle         OREvents;
      EventMaskHandle         NOTEvents;
      struct SleepStruct      **nextSleep;
};
```

The new sleep structure gives you much finer control over what you wish to wake up for. Besides passing the wake up parameters and stay sleeping parameters (the definition of these parameters is determined by the event number) you also pass handles to the events that may relate to the event you are concerned about.

For example, you pass a SleepStruct for a kMonitorMoved event that specifies that you should only be awakened if the monitor moved more that 75 degrees vertically, but stay sleeping if the

move angle exceeds 90 degrees vertical. This may be all that is required, but you may also be concerned about *what* caused that to happen. If you pass an event mask for a `kCatJumpedOnMontior` as one of the `ANDEvent` parameters, then you will be wakened if the 75-90 tilt is the result of the `kCatJumpedOnMontior`. If there are some simultaneous events that you *don't* care about, pass them in the `NOTEvents`. In this case, you may pass a `kEarthQuakeEvent` mask with a value of `kLessThanRichter4.0` as a parameter. This would indicate that you want to be wakened *if* monitor moved more that 75 degrees vertically, but stay sleeping if the move angle exceeds 75 degrees vertical *and* this was not caused by a small earthquake.

A few experiments will make this clear, and you'll be glad to have the control you have.

The next new parameter is the `GestaltAvailableHandle`, this will return to you a list of current system features. This will allow you to dispatch rapidly to the appropriate routine when the user adds or deletes a system feature.

```
struct GestaltAvailable {
      Boolean           changed;
      Boolean           added;
      FeatureStruct     **addedFeatures;
      Boolean           removed;
      FeatureStruct     **removedFeatures;
};
```

where FeatureStruct is:

```
struct FeatureStruct{
      OSType                  selector;
      long                    response;
      OSErr                   result;
      struct FeatureStruct    **nextFeature;
};
```

Review
Draft

The `selector` is self-explanatory. `Response` and `result` are included here, because `GWNE` will automatically call all the currently installed Getstalt selectors during its call.

The next parameter to GWNE is another `GestaltAvailableHandle`. This record specifies the minimum requirements your application has to be awakened again.

While we hope every application is rewritten to take advantage of every possible system configuration dynamically, we understand that there are some smaller shops where this will not be possible for a few months after GWNE goes into general use. For example, there may be some applications that will take a while to revise to continue working when the user removes QuickDraw from the system.

If this is the case for your application, in this parameter all the features that you need to run in `minimumNeeded`.

Note: `Please do not abuse this feature`. If your application is too picky and not ready to handle many different configurations, it is possible for you to call GWNE and never return. The user would be confused by this.

The final new structure is the `GWNECallbackHandle`

```
struct GWNECallbackHandle{
      VoidProcPtr callBack;
      FeatureStruct     **featuresNeeded;
```

```
        short minimumCallBackMinutes;
};
```

Because of the power of GWNE , it sometimes takes a longer time to complete than the older WNE routine. If you would like to take some periodic action during a GWNE call, pass this structure. GWNE will call your callBack proc when the amount of minutes specified in `minimumCallBackMinutes` has elapsed *if* the feature set you defined in `featuresNeeded` is available.

Cautionary Notes

Obviously `GWNE` is going to take a little more time than the older `WaitNextEvent` call. Also, `GWNE` disables interrupts for the duration of the call to prevent new selectors and features from being added while the call is in progress.

This should not be a problem for a well-behaved application, if you are checking Ticks instead of incrementing a variable during interrupt time you will not be affected

Note: `TickCount` now returns minutes, not sixtieths of a second.

We have determined the text editing applications may experience difficulty blinking an insertion point if the user has a great many features installed. We cannot fix this in current System Software, but all new hardware projects will be designed with a 'LCD Shutter' over the display, cycling once every 1.3 seconds. This will simulate the effect of a blinking cursor by blinking the whole screen regularly.

Review

Determining if GWNE is available

At this writing, `GWNE` is designed to be a system extension, and there are no plans to incorporate it in core system software. Incorporating it in the core software would limit its effectiveness.

This means that determining its availability is problematic. You must call `GWNE` to determine if `GWNE` is available. We recommend the following code:

Draft

```
// Prior to calling GWNE, copy all RAM to disk to allow recovery if call fails

   CopyMachineRAMToDisk();  /* your routine */

// Install a bus error handler.  This will point to the code immediatly after
// the GWNE call
   InstallMyBusError();

// Call GWNE

myEvts=GestaltWaitNextEvent(myMaskHandle,mySleepHandle,returnedFeatureSet,mini
mumFeaturesNeeded,callBackHandle);
   if (didBusError){

// this flag will be set by your bus error handler.  If it is set,then GWNE is
// not currently installed.  Reload memory from disk
   CopyDiskImageBackToRAM();  /* your routine */

       CallWaitNextEvent();   // default to calling WNE
   }
```

NOTE: You cannot assume that GWNE will never be available if it was not available one time. The user may install or remove it at any time, so you must write your event loop in this fashion.

Conclusion:

GestaltWaitNextEvent answers the prayers of developers, and the needs of users. It gives a well defined, consistent interface to a fluid environment.

Obviously, existing applications will need some rewriting to become fully GWNE aware. We expect incorporation will take up to two weeks, and re-writing your code to be 'any feature aware' may take slightly longer. However, it will be worth the effort.

Further Reference:

- *Inside Macintosh*, Volume VII-XXIII, Possible Event Codes References

Review Draft

#32: Reserved Resource Types

See: The Resource Manager

Written by: Scott Knaster May 13, 1985
Updated: March 1, 1988

Your applications and desk accessories can create their own resource types. To avoid using type names which have been or will be used in the system, Apple has reserved all resource type names which consist entirely of spaces ($20), lower-case letters ($61 through $7A), and "international" characters (greater than $7F).

In addition Apple has reserved a number of resource types which contain upper-case letters and the "#" character. For a list of these resource types, see The Resource Manager Chapter of *Inside Macintosh* (starting with *Volume V*).

#33: ImageWriter II Paper Motion

Written by: Ginger Jernigan
Updated:

April 30, 1986
March 1, 1988

The purpose of this technical note is to answer the many questions asked about why the paper moves the way it does on the ImageWriter II.

Many people have asked why the paper is rolled backward at the beginning of a Macintosh print job on the ImageWriter II. First, note that this only happens with pin-feed paper (i.e. not with hand-feed or the sheet-feeder) and only at the beginning of a job.

It is not a bug, and it is not malicious programming. It is simply that users are told in the manual to load pin-feed paper with the top edge at the pinch-rollers, making it easy to rip off the printed page(s) without wrecking the paper that is still in the printer or having to roll the paper up and down manually. At the end of every job, the software makes sure that the paper is left in this position, leaving the print-head roughly an inch from the edge. If something is to be printed higher than that, the paper has to be rolled backwards.

As you are probably aware, the "printable rectangle" (`rPage`) reported to the application by the print code begins 1/2 inch from the top edge, not one inch. The reason for that is that we want a document to print exactly the same way whether you are printing on the ImageWriter I or II. On the ImageWriter I, the paper starts with the print-head 1/2 inch from the top edge, so the top of `rPage` is at that position for both printers.

There is no way to eliminate the reverse-feed action, because the user would have to load the paper a different way AND the software would have to know that this was done.

Incidentally, in addition to the paper motion described above, there is also the "burp." This is a 1/8-inch motion back and forth to take up the slop in the printer's gear-train. It is needed on the old-model printer, and there is debate about whether or not it's needed on ALL ImageWriter IIs, or only some, or none. The burp has been in and out of the ImageWriter II code in various releases; right now it's in.

Macintosh Technical Notes

#34: User Items in Dialogs

See also: *Inside Macintosh*, The Dialog Manager

Written by:	Bryan Stearns	May 29, 1985
Updated:		March 1, 1988
Revised by:	Jim Reekes	October 1, 1988

The Dialog Manager does not go into detail about how to manage user items in dialogs; this Technical Note describes the process.
Changes since March 1, 1988: Added MPW C 3.0 code, added a `_SetPort` call to the Pascal example, and noted the necessity and meaning of `enabled` items.

To use a `userItem` with the Dialog Manager, you must define a dialog, load the dialog and install your `userItem`, and respond to events which relate to your `userItem`. If your application wants to receive mouse clicks in the `userItem`, then you must set the item to `enabled`.

Defining a Dialog Box with a userItem

You should define the dialog box in your resource file as follows. Note that it is defined as invisible, since we have to play with the `userItem` before we can draw it.

```
resource 'DLOG' (1001) {                    /* type/ID for box */
   {100,100,300,400},                       /* rectangle for window */
   dBoxProc, invisible, noGoAway, 0x0,      /* note it is invisible */
   1001,
   "Test Dialog"
};

resource 'DITL' (1001) {                    /* matching item list */
   {
      {160, 190, 180, 280},                 /* rectangle for button */
         button { enabled, "OK" };          /* an OK button */
      {104, 144, 120, 296},                 /* rectangle for item */
         userItem { enabled }               /* a user item! */
   }
};
```

Loading and Preparing to Show the Dialog Box

Before we can actually show the dialog box to the user, we need two support routines. The Dialog Manager calls the first procedure whenever we need to draw our `userItem`. You should install it (as shown below) after calling `_GetNewDialog` but before calling `_ShowWindow`. This first procedure simply draws the `userItem`.

In MPW Pascal:

```
PROCEDURE MyDraw(theDialog: DialogPtr; theItem: INTEGER);

        VAR
           iType : INTEGER;                    {returned item type}
           iBox  : Rect;                       {returned bounds rect}
           iHdl  : Handle;                     {returned item handle}

        BEGIN
           GetDItem(theDialog,theItem,iType,iHdl,iBox); {get the box}
           FillRect(iBox,ltGray);              {fill with light gray}
           FrameRect(iBox);                    {frame it}
        END; {MyDraw}
```

In MPW C 3.0:

```
      pascal void MyDraw(theDialog,theItem)
      DialogPtr     theDialog;
      short int     theItem;

      {
            short int     iType;              /*returned item type*/
            Rect          iBox;               /*returned bounds rect*/
            Handle        iHdl;               /*returned item handle*/

            GetDItem(theDialog,theItem,&iType,&iHdl,&iBox); /*get the box*/
            FillRect(&iBox,qd.ltGray);        /*fill with light gray*/
            FrameRect(&iBox);                 /*frame it*/
      } /*MyDraw*/
```

The other necessary procedure is a filter procedure (`filterProc`) that the Dialog Manager calls whenever `_ModalDialog` receives an event (this only applies when calling `_ModalDialog`; modeless dialogs are covered below). The default `filterProc` looks for key-down and auto-key events and simulates pressing the OK button (or whatever else is item 1) if the user has pressed either the Return key or the Enter key. To support a `userItem`, the `filterProc` must handle events for any `userItem` items in the dialog in addition to performing the default `filterProc` tasks. The following short `filterProc` supports these types of items; when the user clicks in the `userItem`, the `filterProc` inverts it.

In MPW Pascal:

```
      FUNCTION MyFilter(theDialog: DialogPtr; VAR theEvent: EventRecord;
                        VAR itemHit: INTEGER): BOOLEAN;
         CONST
            enterKey    = 3;
            returnKey   = 13;

         VAR
            mouseLoc : Point;                 {we'll play w/ mouse}
            key      : SignedByte;            {for enter/return}
            iBox     : Rect;                  {returned boundsrect}
            iHdl     : Handle;                {returned item handle}
            iType, itemHit : INTEGER;         {returned item and type}

         BEGIN
            SetPort(theDialog);
            MyFilter := FALSE;                {assume not our event}
```

```
      CASE theEvent.what OF                  {which event?}
         keyDown,autoKey: BEGIN              {he hit a key}
            key := SignedByte(event.message); {get keycode}
            IF (key = enterKey) OR (key = returnKey ) THEN BEGIN
               MyFilter := TRUE;             {we handled it}
               itemHit := 1;                 {he hit the 1st item}
            END;                             {test CR or Enter}
         END;                                {keydown}
         mouseDown: BEGIN                    {he clicked}
            mouseLoc := theEvent.where;      {get the mouse pos'n}
            GlobalToLocal(mouseLoc);         {convert to local}
            GetDItem(theDialog,2,iType,iHdl,iBox); {get our box}
            IF PtInRect(mouseLoc,iBox) THEN BEGIN {he hit our item}
               InvertRect(iBox);
               MyFilter := TRUE;             {we handled it}
               itemHit := 2;                 {he hit the userItem}
            END;                             {if he hit our userItem}
         END;                                {mousedown}
      END;                                   {event case}
   END;                                      {MyFilter}
```

In MPW C 3.0:

```
pascal Boolean MyFilter(theDialog,theEvent,itemHit)
DialogPtr      theDialog;
EventRecord    *theEvent;
short int      *itemHit;

#define enterKey      3;                     /*the enter key*/
#define returnKey     13;                    /*the return key*/

{
       char           key;                   /*for enter/return*/
       short int      iType;                 /*returned item type*/
       Rect           iBox;                  /*returned boundsrect*/
       Handle         iHdl;                  /*returned item handle*/
       Point          mouseLoc;              /*we'll play w/ mouse*/

       SetPort(theDialog);
       switch (theEvent->what)                /*which event?*/
       {

              case keyDown:
              case autoKey: /*he hit a key*/
                     key = theEvent->message; /*get ascii code*/
                     if ((key == enterKey) || (key == returnKey))
                     {                      /*he hit CR or Enter*/
                            *itemHit = 1;  /*he hit the 1st item*/
                            return(true);  /*we handled it*/
                     } /*he hit CR or enter*/
                     break;                 /* case keydown, case autoKey */
              case mouseDown:               /*he clicked*/
                     mouseLoc = theEvent->where;  /*get the mouse pos'n*/
                     GlobalToLocal(&mouseLoc);    /*convert to local*/
                     GetDItem(theDialog,2,&iType,&iHdl,&iBox); /*get our box*/
                     if (PtInRect(mouseLoc,&iBox))
                     {                      /*he hit our item*/
                            InvertRect(&iBox);
                            *itemHit = 2;  /*he hit the userItem*/
                            return(true);  /*we handled it*/
                     } /*if he hit our userItem*/
                     break; /*case mouseDown */
       } /*event switch*/
       return(false); /* we're still here, so return false
                            (we didn't handle the event) */
} /*MyFilter*/
```

Invoking the Dialog Box

When we need this dialog box, we load it into memory as follows:

In MPW Pascal:

```
PROCEDURE DoOurDialog;

   VAR
      myDialog : DialogPtr;                {the dialog pointer}
      iType, itemHit : INTEGER;            {returned item type}
      iBox     : Rect;                     {returned boundsRect}
      iHdl     : Handle;                   {returned item Handle}

   BEGIN
      myDialog := GetNewDialog(1001,nil,POINTER(-1)); {get the box}
      GetDItem(myDialog,2,iType,iHdl,iBox); {2 is the item number}
      SetDItem(myDialog,2,iType,@myDraw,iBox); {install draw proc}
      ShowWindow(theDialog);               {make it visible}
      REPEAT
         ModalDialog(@MyFilter, itemHit ); {let dialog manager run it}
      UNTIL itemHit = 1;                   {until he hits ok.}
      DisposDialog(myDialog);              {throw it away}
   END;                                    {DoOurDialog}
```

In MPW C 3.0:

```
void DoOurDialog()

{
       DialogPtr      myDialog;      /*the dialog pointer*/
       short int      iType;         /*returned item type*/
       short int      itemHit;       /*returned from ModalDialog*/
       Rect           iBox;          /*returned boundsRect*/
       Handle         iHdl;          /*returned item Handle*/

       myDialog = GetNewDialog(1001,nil,(WindowPtr)-1); /*get the box*/
       GetDItem(myDialog,2,&iType,&iHdl,&iBox); /*2 is the item number*/
       SetDItem(myDialog,2,iType,MyDraw,&iBox); /*install draw proc*/
       ShowWindow(myDialog);          /*make it visible*/

       while (itemHit != 1) ModalDialog(MyFilter, &itemHit);
       DisposDialog(myDialog);        /*throw it away*/
}                                     /*DoOurDialog*/
```

Using userItem Items with Modeless Dialogs

If you are using `userItem` items in modeless dialog box, the Dialog Manager will call the draw procedure when `_DialogSelect` receives an update event for the dialog box. When the user clicks on your `userItem` and it is `enabled`, `_DialogSelect` will return `TRUE`. The `itemHit` will be equal to the item number of your `userItem`. Your code can then handle this like the mouse-down event case in the example above.



#35: DrawPicture Problem

Written by: Mark Baumwell
Updated:

June 19, 1986
March 1, 1988

This note formerly described a problem with DrawPicture that occurred only on 64K ROM machines. Information specific to 64K ROM machines has been deleted from Macintosh Technical Notes for reasons of clarity.

#36: Drive Queue Elements

See also: The File Manager
The Device Manager

Written by: Bryan Stearns June 12, 1985
Updated: March 1, 1988

This note expands on *Inside Macintosh*'s definition of the drive queue, which is given in the File Manager chapter.

As shown in *Inside Macintosh*, a drive queue element has the following structure:

```
DrvQEl = RECORD
    qLink:    QElemPtr; {next queue entry}
    qType:    INTEGER;  {queue type}
    dQDrive:  INTEGER;  {drive number}
    dQRefNum: INTEGER;  {driver reference number}
    dQFSID:   INTEGER;  {file-system identifier}
    dQDrvSz:  INTEGER;  {number of logical blocks on drive}
    dQDrvSz2: INTEGER;  {additional field to handle large drive size}
END;
```

Note that dQDrvSz2 is only used if qType is 1. In this case, dQDrvSz2 contains the high-order word of the size, and dQDrvSz contains the low-order word.

Inside Macintosh also mentions four bytes of flags that preced each drive queue entry. How are these flags accessed? The flags begin 4 bytes before the address pointed to by the `DrvQElPtr`. In assembly language, accessing this isn't a problem:

```
MOVE.L  -4(A0),D0   ;A0 = DrvQElPtr; get drive queue flags
```

If you're using Pascal, it's a little more complicated. You can get to the `flags` with this routine:

```
FUNCTION DriveFlags(aDQEPtr: DrvQElPtr): LONGINT;

   VAR
      flagsPtr : ^LONGINT; {we'll point at drive queue flags with this}

   BEGIN
      {subtract 4 from the DrvQElPtr, and get the LONGINT there}
      flagsPtr := POINTER(ORD4(aDQEPtr) - 4);
      DriveFlags := flagsPtr^;
   END;
```

From MPW C, you can use:

```
long DriveFlags(aDQEPtr)
DrvQElPtr     aDQEPtr;

{ /* DriveFlags */
    return(*((long *)aDQEPtr - 1));  /* coerce flagsPtr to a (long *)
                                        so that subtracting 1 from it
                                        will back us up 4 bytes */
} /* DriveFlags */
```

Creating New Drives

To add a drive to the drive queue, assembly-language programmers can use the function defined below. It takes two parameters: the driver reference number of the driver which is to "own" this drive, and the size of the new drive in blocks. It returns the drive number created. It is vital that you **not** hard-code the drive number; if the user has installed other non-standard drives in the queue, the drive number you're expecting may already be taken. (Note that the example function below arbitrates to find an unused drive number, taking care of this problem for you. Also, note that this function doesn't mount the new volume; your code should take care of that, calling the Disk Initialization Package to reformat the volume if necessary).

```
AddMyDrive  PROC     EXPORT
;-------------------------------------------------------------------------
;FUNCTION AddMyDrive(drvSize: LONGINT; drvrRef: INTEGER): INTEGER;
;-------------------------------------------------------------------------
;Add a drive to the drive queue. Returns the new drive number, or a negative
;error code (from trying to allocate the memory for the queue element).
;-------------------------------------------------------------------------
DQESize     EQU      18          ;size of a drive queue element
;We use a constant here because the number in SysEqu.a doesn't take into
;account the flags LONGINT before the element, or the size word at the end.
;-------------------------------------------------------------------------
StackFrame  RECORD   {link},DECR
result      DS.W     1           ;function result
params      EQU      *
drvSize     DS.L     1           ;drive size parameter
drvrRef     DS.W     1           ;drive refNum parameter
paramSize   EQU      params-*
return      DS.L     1           ;return address
link        DS.L     1           ;saved value of A6 from LINK
block       DS.B     ioQElSize   ;parameter block for call to MountVol
linkSize    EQU      *
            ENDR
;-------------------------------------------------------------------------
            WITH     StackFrame     ;use the offsets declared above

            LINK     A6,#linkSize   ;create stack frame

            ;search existing drive queue for an unused number

            LEA      DrvQHdr,A0     ;get the drive queue header
            MOVEQ    #4,D0          ;start with drive number 4
```

```
CheckDrvNum
        MOVE.L   qHead(A0),A1    ;start with first drive
CheckDrv
        CMP.W    dqDrive(A1),D0 ;does this drive already have our number?
        BEQ.S    NextDrvNum     ;yep, bump the number and try again.
        CMP.L    A1,qTail(A0)   ;no, are we at the end of the queue?
        BEQ.S    GotDrvNum      ;if yes, our number's unique! Go use it.
        MOVE.L   qLink(A1),A1   ;point to next queue element
        BRA.S    CheckDrv       ;go check it.

NextDrvNum
        ;this drive number is taken, pick another

        ADDQ.W   #1,D0          ;bump to next possible drive number
        BRA.S    CheckDrvNum    ;try the new number

GotDrvNum
        ;we got a good number (in D0.W), set it aside

        MOVE.W   D0,result(A6)  ;return it to the user

        ;get room for the new DQE

        MOVEQ    #DQESize,D0    ;size of drive queue element, adjusted
        _NewPtr  sys            ;get memory for it
        BEQ.S    GotDQE         ;no error...continue
        MOVE.W   D0,result(A6)  ;couldn't get the memory! return error
        BRA.S    FinishUp       ;and exit

GotDQE
        ;fill out the DQE

        MOVE.L   #$80000,(A0)+  ;flags: non-ejectable; bump past flags

        MOVE.W   #1,qType(A0)   ;qType of 1 means we do use dQDrvSz2
        CLR.W    dQFSID(A0)     ;"local file system"
        MOVE.W   drvSize(A6),dQDrvSz2(A0)  ;high word of number of blocks
        MOVE.W   drvSize+2(A6),dQDrvSz(A0) ;low word of number of blocks

        ;call AddDrive

        MOVE.W   result(A6),D0  ;get the drive number back
        SWAP     D0             ;put it in the high word
        MOVE.W   drvrRef(A6),D0 ;move the driver refNum in the low word
        _AddDrive               ;add this drive to the drive queue

FinishUp
        UNLK     A6             ;get rid of stack frame
        MOVE.L   (SP)+,A0       ;get return address
        ADDQ     #paramSize,SP  ;get rid of parameters
        JMP      (A0)           ;back to caller
;---------------------------------------------------------------------------
        ENDPROC
```

#37: Differentiating Between Logic Boards

See: Technical Note #129—SysEnvirons

Written by: Mark Baumwell June 19, 1986
Updated: March 1, 1988

Earlier versions of this note are obsoleted by existence of `SysEnvirons`, which is documented in Technical Note #129.

#38: The ROM Debugger

Written by: Louella Pizzuti
Updated:

June 20, 1986
March 1, 1988

The debugger in ROM (not present on the Macintosh 128, Macintosh 512, or Macintosh XL) recognizes the following commands:

`PC [expr]` (program counter)

Typing `PC` on a line by itself displays the program counter. Typing `PC 50000` sets the program counter to `$50000`.

`SM [address [number(s)]]` (set memory)

Typing `SM` on a line by itself displays the next 96 bytes of memory. Typing `SM 50000` will display memory starting at `$50000`. Typing `SM 50000 4849 2054 6865 7265 2120` will set memory starting at `$50000` to `$4849`... Subsequently hitting Return will increment the display a screen at a time.

`DM [address]` (display memory)

Typing `DM` on a line by itself displays the next 96 bytes of memory. Typing `DM 50000` will display memory at `$50000`. Subsequently hitting Return will increment the display a screen at a time.

`SR [expr]` (status register)

Typing `SR` on a line by itself displays the status register. Typing `SR 2004` sets the status register to `$2004`.

`TD` (total display)

Displays memory at the "magic" location `$3FFC80`, which contains the current values of the registers. The registers are displayed in the following order: `D0-D7, A0-A7, PC, SR`.

`G [address]` (go)

Executes instructions starting at address. If `G` is typed on a line by itself, execution begins at the address indicated by the program counter.

Note: If you want to exit to the shell, you just need to type: `SM 0 A9F4`, then `G 0`

Note: If you crash into the debugger and the system hangs, try turning off your modem.

#39: Segment Loader Patch

Written by:	Russ Daniels Bryan Stearns	August 1, 1985
Modified by:	Jim Friedlander	November 15, 1986
Updated:		March 1, 1988

This note formerly described a patch to the Segment Loader for 64K ROM machines. Information specific to 64K ROM machines has been deleted from Macintosh Technical Notes for reasons of clarity.

#40: Finder Flags

See also: The File Manager

Written by: Jim Friedlander June 16, 1986
Modified by: Jim Friedlander March 2, 1987
Updated: March 1, 1988

This revision corrects the meanings of bits 6 and 7, which were interchanged in the older version of this technical note. ResEdit uses these bits incorrectly in versions older than 1.2.

The Finder keeps and uses a series of file information flags for each file. These flags are located in the `fdFlags` field (a word at offset `$28` into an `HParamBlockRec`) of the `ioFlFndrInfo` record of a parameter block. They may change with newer versions of the Finder. Finders 5.4 and newer assign the following meanings to the flags:

Bit	Meaning
0	Set if file/folder is on the desktop (Finder 5.0 and later)
1	bFOwnAppl (used internally)
2	reserved (currently unused)
3	reserved (currently unused)
4	bFNever (never SwitchLaunch) (not implemented)
5	bFAlways (always SwitchLaunch)
6	Set if file is a shareable application
7	reserved (used by System)
8	Inited (seen by Finder)
9	Changed (used internally by Finder)
10	Busy (copied from File System busy bit)
11	NoCopy (not used in 5.0 and later, formerly called BOZO)
12	System (set if file is a system file)
13	HasBundle
14	Invisible
15	Locked

Macintosh
Technical Notes

Developer Technical Support

#41: Drawing Into an Off-Screen Bitmap

Revised by: Jon Zap & Forrest Tanaka June 1990
Written by: Jim Friedlander & Ginger Jernigan July 1985

This Technical Note provides an example of creating an off-screen bitmap, drawing to it, and then copying from it to the screen.
Changes since April 1990: Clarified the section on window updates with off-screen bitmaps to explicitly limit these updates to your own windows.

The following is an example of creating and drawing to an off-screen bitmap, then copying from it to an on-screen window. We supply this example in both MPW Pascal and C.

MPW Pascal

First, let's look at a general purpose function to create an off-screen bitmap. This function creates the `GrafPort` on the heap. You could also create it on the stack and pass the uninitialized structure to a function similar to this one.

```
FUNCTION CreateOffscreenBitMap(VAR newOffscreen:GrafPtr; inBounds:Rect) : BOOLEAN;

VAR
  savePort  : GrafPtr;
  newPort   : GrafPtr;

BEGIN
  GetPort(savePort);        {need this to restore thePort after OpenPort changes it}

  newPort := GrafPtr(NewPtr(sizeof(GrafPort)));     {allocate the GrafPort}
  IF MemError <> noErr THEN BEGIN
    CreateOffscreenBitMap := false;                 {failed to allocate it}
    EXIT(CreateOffscreenBitMap);
  END;
  {
  the OpenPort call does the following . . .
    allocates space for visRgn (set to screenBits.bounds) and clipRgn (set wide open)
    sets portBits to screenBits
    sets portRect to screenBits.bounds
    etc. (see IM I-163,164)
    side effect: does a SetPort(offScreen)
  }
  OpenPort(newPort);
  {make bitmap exactly the size of the bounds that caller supplied}
  WITH newPort^ DO BEGIN {portRect, clipRgn, and visRgn are in newPort}
    portRect := inBounds;
    RectRgn(clipRgn, inBounds);         {avoid wide-open clipRgn, to be safe}
    RectRgn(visRgn, inBounds);          {in case inBounds is > screen bounds}
  END;
```

```
  WITH newPort^.portBits DO BEGIN       {baseAddr, rowBytes and bounds are in newPort}
    bounds := inBounds;
    {rowBytes is size of row  It must be rounded up to even number of bytes}
    rowBytes := ((inBounds.right - inBounds.left + 15) DIV 16) * 2;

    {number of bytes in BitMap is rowBytes * number of rows}
    {see note at end of Technical Note about using _NewHandle rather than _NewPtr}
    baseAddr := NewPtr(rowBytes * LONGINT(inBounds.bottom - inBounds.top));
  END;
  IF MemError <> noErr THEN BEGIN    {see if we had enough room for the bits}
    SetPort(savePort);
    ClosePort(newPort);              { dump the visRgn and clipRgn }
    DisposPtr(Ptr(newPort));         { dump the GrafPort}
    CreateOffscreenBitMap := false;
  END
  ELSE BEGIN
    { since the bits are just memory, let's erase them before we start }
    EraseRect(inBounds);            {OpenPort did a SetPort(newPort)}
    newOffscreen := newPort;
    SetPort(savePort);
    CreateOffscreenBitMap := true;
  END;
END;
```

Here is the procedure to get rid of an off-screen bitmap created by the previous function:

```
PROCEDURE DestroyOffscreenBitMap(oldOffscreen : GrafPtr);
BEGIN
  ClosePort(oldOffscreen);                        { dump the visRgn and clipRgn }
  DisposPtr(oldOffscreen^.portBits.baseAddr);     { dump the bits }
  DisposPtr(Ptr(oldOffscreen));                   { dump the port }
END;
```

Now that you know how to create and destroy an off-screen bitmap, let's go through the motions of using one. First, let's define a few things to make the `_NewWindow` call a little clearer.

```
CONST
  kIsVisible    = true;
  kNoGoAway     = false;
  kMakeFrontWindow = -1;
  myString      = 'The EYE';  {string to display}
```

Here's the body of the test code:

```
VAR
  offscreen : GrafPtr;     {our off-screen bitmap}
  ovalRect  : Rect;        {used for example drawing}
  myWBounds : Rect;        {for creating window}
  OSRect    : Rect;        {portRect and bounds for off-screen bitmap}
  myWindow  : WindowPtr;

BEGIN
  InitToolbox;                          {exercise left to the reader}

  myWBounds := screenBits.bounds;     { size of main screen }
  InsetRect(myWBounds, 50,50);        { make it fit better }
  myWindow := NewWindow(NIL, myWBounds, 'Test Window', kIsVisible,
                        noGrowDocProc, WindowPtr(kMakeFrontWindow), kNoGoAway, 0);

  IF NOT CreateOffscreenBitMap(offscreen,myWindow^.portRect) THEN BEGIN
    SysBeep(1);
    ExitToShell;
  END;
```

```
  { Example drawing to our off-screen bitmap }
  SetPort(offscreen);
  OSRect := offscreen^.portRect;    { offscreen bitmap's local coordinate rect }
  ovalRect := OSRect;
  FillOval(ovalRect, black);
  InsetRect(ovalRect, 1, 20);
  FillOval(ovalRect, white);
  InsetRect(ovalRect, 40, 1);
  FillOval(ovalRect, black);
  WITH ovalRect DO
    MoveTo((left+right-StringWidth(myString)) DIV 2, (top+bottom-12) DIV 2);
  TextMode(srcXor);
  DrawString(myString);

  { copy from the off-screen bitmap to the on-screen window.  Note that in this
  case the source and destination rects are the same size and both cover the
  entire area.  These rects are allowed to be portions of the source and/or
  destination and do not have to be the same size.  If they are not the same size
  then _CopyBits scales the image accordingly
  }
  SetPort(myWindow);
  CopyBits(offscreen^.portBits, myWindow^.portBits,
           offscreen^.portRect, myWindow^.portRect, srcCopy, NIL);

  DestroyOffscreenBitMap(offscreen);    {remove the evidence}

  WHILE NOT Button DO;                  {give user a chance to see the results}
END.
```

MPW C

First, let's look at a general purpose function to create an off-screen bitmap. This function creates the `GrafPort` on the heap. You could also create it on the stack and pass the uninitialized structure to a function similar to this one.

```
Boolean CreateOffscreenBitMap(GrafPtr *newOffscreen, Rect *inBounds)
{
  GrafPtr savePort;
  GrafPtr newPort;

  GetPort(&savePort);    /* need this to restore thePort after OpenPort */

  newPort = (GrafPtr) NewPtr(sizeof(GrafPort));    /* allocate the grafPort */
  if (MemError() != noErr)
    return false;                 /* failed to allocate the off-screen port */
  /*
  the call to OpenPort does the following . . .
    allocates space for visRgn (set to screenBits.bounds) and clipRgn (set wide open)
    sets portBits to screenBits
    sets portRect to screenBits.bounds
    etc. (see IM I-163,164)
    side effect: does a SetPort(&offScreen)
  */
  OpenPort(newPort);
  /* make bitmap the size of the bounds that caller supplied */
  newPort->portRect = *inBounds;
  newPort->portBits.bounds = *inBounds;
  RectRgn(newPort->clipRgn, inBounds);    /* avoid wide-open clipRgn, to be safe  */
  RectRgn(newPort->visRgn, inBounds);     /* in case newBounds is > screen bounds */

  /* rowBytes is size of row, it must be rounded up to an even number of bytes */
  newPort->portBits.rowBytes = ((inBounds->right - inBounds->left + 15) >> 4) << 1;
```

```
  /* number of bytes in BitMap is rowBytes * number of rows */
  /* see notes at end of Technical Note about using _NewHandle rather than _NewPtr */
  newPort->portBits.baseAddr =
          NewPtr(newPort->portBits.rowBytes * (long) (inBounds->bottom - inBounds->top));
  if (MemError()!=noErr) {    /* check to see if we had enough room for the bits */
    SetPort(savePort);
    ClosePort(newPort);       /* dump the visRgn and clipRgn */
    DisposPtr((Ptr)newPort); /* dump the GrafPort */
    return false;             /* tell caller we failed */
    }
  /* since the bits are just memory, let's clear them before we start */
  EraseRect(inBounds);      /* OpenPort did a SetPort(newPort) so we are ok */
  *newOffscreen = newPort;
  SetPort(savePort);
  return true;                /* tell caller we succeeded! */
}
```

Here is the function to get rid of an off-screen bitmap created by the previous function:

```
void DestroyOffscreenBitMap(GrafPtr oldOffscreen)
{
  ClosePort(oldOffscreen);                        /* dump the visRgn and clipRgn */
  DisposPtr(oldOffscreen->portBits.baseAddr);     /* dump the bits */
  DisposPtr((Ptr)oldOffscreen);                   /* dump the port */
}
```

Now that you know how to create and destroy an off-screen bitmap, let's go through the motions of using one. First, let's define a few things to make the _NewWindow call a little clearer.

```
#define kIsVisible true
#define kNoGoAway false
#define kNoWindowStorage 0L
#define kFrontWindow ((WindowPtr) -1L)
```

Here's the body of the test code:

```
main()
{
  char* myString = "\pThe EYE";  /* string to display */

  GrafPtr   offscreen;          /* our off-screen bitmap */
  Rect      ovalRect;           /* used for example drawing */
  Rect      myWBounds;          /* for creating window */
  Rect      OSRect;             /* portRect and bounds for off-screen bitmap*/
  WindowPtr myWindow;

  InitToolbox();                /* exercise for the reader */
  myWBounds = qd.screenBits.bounds;  /* size of main screen */
  InsetRect(&myWBounds, 50,50);  /* make it fit better */
  myWindow = NewWindow(kNoWindowStorage, &myWBounds, "\pTest Window", kIsVisible,
                       noGrowDocProc, kFrontWindow, kNoGoAway, 0);
  if (!CreateOffscreenBitMap(&offscreen, &myWindow->portRect)) {
    SysBeep(1);
    ExitToShell();
    }
```

```
  /* Example drawing to our off-screen bitmap*/
  SetPort(offscreen);
  OSRect = offscreen->portRect;  /* offscreen bitmap's local coordinate rect */
  ovalRect = OSRect;
  FillOval(&ovalRect, qd.black);
  InsetRect(&ovalRect, 1, 20);
  FillOval(&ovalRect, qd.white);
  InsetRect(&ovalRect, 40, 1);
  FillOval(&ovalRect, qd.black);
  MoveTo((ovalRect.left + ovalRect.right - StringWidth(myString)) >> 1,
         (ovalRect.top + ovalRect.bottom - 12) >> 1);
  TextMode(srcXor);
  DrawString(myString);

  /* copy from the off-screen bitmap to the on-screen window.  Note that in this
  case the source and destination rects are the same size and both cover the
  entire area.  These rects are allowed to be portions of the source and/or
  destination and do not have to be the same size.  If they are not the same size
  then _CopyBits scales the image accordingly.
  */
  SetPort(myWindow);
  CopyBits(&offscreen->portBits, &(*myWindow).portBits,
           &offscreen->portRect, &(*myWindow).portRect, srcCopy, 0L);

  DestroyOffscreenBitMap(offscreen);     /* dump the off-screen bitmap */
  while (!Button());      /* give user a chance to see our work of art */
}
```

Comments

In the example code, the bits of the `BitMap` structure, which are pointed to by the `baseAddr` field, are allocated by a `_NewPtr` call. If your off-screen bitmap is close to the size of the screen, then the amount of memory needed for the bits can be quite large (on the order of 20K for the Macintosh SE or 128K for a large screen). This is quite a lot of memory to lock down in your heap and it can easily lead to fragmentation if you intend to keep the off-screen bitmap around for any length of time. One alternative that lessens this problem is to get the bits via `_NewHandle` so the Memory Manager can move them when necessary. To implement this approach, you need to keep the handle separate from the `GrafPort` (for example, in a structure that combines a `GrafPort` and a `Handle`). When you want to use the off-screen bitmap you would then lock the handle and put the dereferenced handle into the `baseAddr` field. When you are not using the off-screen bitmap you can then unlock it.

This example does not demonstrate one of the more typical uses of off-screen bitmaps, which is to preserve the contents of windows so that after a temporary window or dialog box obscures part of your windows and is then dismissed, you can quickly handle the resulting update events without recreating all of the intermediate drawing commands.

Make sure you only restore the pixels within the content regions of your own windows in case the temporary window partly obscures windows belonging to other applications or to the desktop. Another application could change the contents of its windows while they are behind your temporary window, so you cannot simply restore all the pixels that were behind the temporary window because that would restore the old contents of the other application's windows. Instead, you could keep keep an off-screen bitmap for each of your windows and then restore them by copying each bit map into the corresponding window's ports when they get their update events.

An alternate method is to make a single off-screen bitmap that is as large as the temporary window and a region that is the union of the content regions of your windows. Before you display the

temporary window, copy the screen into the off-screen bit map using the region as a mask. After the temporary window is dismissed, restore the obscured area by copying from the off-screen bit map into a copy of the Window Manager port, and use the region as a mask. If the region has the proper shape and location, it prevents `_CopyBits` from drawing outside of the content regions of your windows. See Technical Note #194, WMgrPortability for details about drawing across windows.

In some cases it can be just as fast and convenient to simply define a picture (PICT) and then draw it into your window when necessary. There are cases, however, such as text rotation, where it is advantageous to do the drawing off the screen, manipulate the bit image, and then copy the result to the visible window (thus avoiding the dangers inherent in writing directly to the screen). In addition, this technique reduces flicker, because all of the drawing done off the screen appears on the screen at once.

It is also important to realize that, if you plan on using the pre-Color QuickDraw eight-color model, an off-screen bitmap loses any color information and you do not see your colors on a system that is capable of displaying them. In this case you should either use a PICT to save the drawing information or check for the presence of Color QuickDraw and, when it is present, use a `PixMap` instead of a `BitMap` and the color toolbox calls (*Inside Macintosh*, Volume V) instead of the standard QuickDraw calls (*Inside Macintosh*, Volume I).

You may also want to refer to the OffScreen library (DTS Sample Code #15) which provides both high- and low-level off-screen bitmap support for the 128K and later ROMs. The OffSample application (DTS Sample Code #16) demonstrates the use of this library.

Further Reference:

- *Inside Macintosh*, Volumes I & IV, QuickDraw
- *Inside Macintosh*, Volume V, Color QuickDraw
- Technical Note #120, Drawing Into an Off-Screen Pixel Map
- Technical Note #194, WMgrPortability
- DTS Macintosh Sample Code #15, OffScreen & #16, OffSample

#42: Pascal Routines Passed by Pointer

See also: Macintosh Memory Management: An Introduction

Written by: Scott Knaster July 22, 1985
Updated: March 1, 1988

Routines passed by pointer are used in many places in conjunction with Macintosh system routines. For example, filter procedures for modal dialogs are passed by pointer, as are controls' action procedures (when calling `TrackControl`), and I/O completion routines.

If you're using MPW Pascal, the syntax is usually

```
partCode := TrackControl(theControl, startPt, @MyProc)
```

where `MyProc` is the procedure passed by pointer (using the @ symbol).

Because of the way that MPW Pascal (and some other compilers) construct stack frames, any procedure or function passed by pointer **must not** have its declaration nested within another procedure or function. If its declaration is nested, the program will crash, probably with an illegal instruction error. The following example demonstrates this:

```
PROGRAM CertainDeath;

   PROCEDURE CallDialog;

      VAR
         x : INTEGER;

      FUNCTION MyFilter(theDialog: DialogPtr; VAR theEvent: EventRecord;
                        VAR itemHit: INTEGER): Boolean;
      {note that MyFilter's declaration is nested within CallDialog}

      BEGIN {MyFilter}
        {body of MyFilter}
      END; {MyFilter}

   BEGIN {CallDialog}
      ModalDialog(@MyFilter,itemHit) {<------------ will crash here}
   END; {CallDialog}

BEGIN {main program}
      CallDialog;
END.
```

#43: Calling LoadSeg

See also: The Segment Loader

Written by: Gene Pope
Updated:

October 15, 1985
March 1, 1988

Earlier versions of this note described a way to call the `LoadSeg` trap, which is used internally by the Segment Loader. We no longer recommend calling `LoadSeg` directly.

#44: HFS Compatibility

See also: The File Manager

Written by: Jim Friedlander October 9, 1985
Modified by: Scott Knaster December 5, 1985
Jim Friedlander
Updated: March 1, 1988

This technical note tells you how to make sure that your applications run under the Hierarchical File System (HFS).

The Hierarchical File System (HFS) provides fast, efficient management of larger volumes than the original Macintosh File System (MFS). Since HFS is hierarchical, HFS folders have a meaning different from MFS folders. In MFS, a folder has only graphical significance—it is only used by the Finder as a means of visually grouping files. The MFS directory structure is actually flat (all files are at the 'root' level). Under HFS, a folder is a directory that can contain files and other directories.

A folder is accessed by use of a `WDRefNum` (Working Directory reference number). Calls that return a `vRefNum` when running under MFS may return a `WDRefNum` when running under HFS. You may use a `WDRefNum` wherever a `vRefNum` may be used.

In order to provide for compatibility with software written for MFS, the HFS calls that open files search both the default directory and the directory that contains the System and the Finder (HFS marks this last directory so it always knows where to look for the System and the Finder).

Your goal should be to write programs that are file system independent. Your programs should not only be able to access files on other volumes, but also files that are in other directories. Accomplishing this is not difficult—most applications that were written for MFS work correctly under HFS. If you find that your current applications do not run correctly under HFS, you should check to see if you are doing any of the following five things:

Are you using Standard File?

This is very important to ensure that your application will run correctly under HFS. HFS uses an extended Standard File, which allows the user to select from files in different directories. This increased functionality was implemented without changing Standard File's external specification—the only difference is that `SFReply.vRefNum` can now be a `WDRefNum`. Please note that using Standard File's dialog hook and filter procs or adding controls of your own will not cause compatibility problems with HFS.

Existing applications that use Standard File properly run without modification under HFS. Applications that take the `SFReply.vRefNum` and convert that to a volume name, then append it to `SFReply.fName` (as in #2 below) do not function correctly under HFS—the user can only open files in the root directory. If you call `Open` with `SFReply.vRefNum` and `SFReply.fName`, everything will work correctly. Remember, `SFReply.vRefNum` may be a `WDRefNum` . Using Standard File will virtually guarantee that your application will be compatible with MFS, HFS, and future file systems.

Are you concatenating volume names to file names, i.e. using file names of the form `VOLUME:fileName`?

Applications that do this do not work correctly under HFS (in fact, they do not even run correctly under MFS). Instead of this, use a `vRefNum` to access a volume or a directory. Fully qualified pathnames (such as `volume:folder1:folder2:filename`) work correctly, but we don't recommend that you use them. Please don't ever make a user type in a full pathname!

Are you searching directories for files using a loop such as

```
FOR index:= 1 to ioVNmFls DO ...
```

where `ioVNmFls` was returned from a `PBGetVinfo` call?

This technique should not be used. Instead, use repeated calls to `PBGetFInfo` using `ioFDirIndex` until `fnfErr` is returned. Indexed calls to `PBGetFInfo` will return files in the directory specified by the `vRefNum` that you put in the parameter block.

Are you assuming that a `vRefNum` will actually refer to a volume?

A `vRefNum` can now be a `WDRefNum`. A `WDRefNum` indicates which working directory (folder) a file is in, not which volume the file is on. Don't think of a `vRefNum` as a way to access a volume, but rather as a means of telling the file system where to find a file.

Are you walking through the VCB queue?

You should let us do the walking for you. Using indexed calls to `PBGetVInfo` will allow you to get information about any mounted volume. You shouldn't walk through the VCB queue because it changed for HFS and might change in the future. The routines that we supply will correctly access information in the VCB queue.

Are you using the file system's "`IMMED`" bit? (assembly language only)

Inside Macintosh describes bit 9 of the trap word as the immediate bit. In fact, setting this bit under MFS did not work as documented; it did not have the desired effect of bypassing the file I/O queue. Under HFS, this bit is used; it distinguishes HFS varieties of calls from MFS varieties. For example, the `PBOpen` call has this bit clear; `PBHOpen` has it set. Therefore, you must be sure that your file system calls do not use this bit as the immediate bit.

#45: *Inside Macintosh* Quick Reference

Compiled by: Jim Friedlander
August 2, 1985
Updated: March 1, 1988

This note formerly listed the traps from *Inside Macintosh Volumes I-III*. Better references are now available elsewhere.

#46: Separate Resource Files

See also: The Resource Manager

Written by: Bryan Stearns October 16, 1985
Updated: March 1, 1988

During application development, you use a resource compiler (RMaker or Rez) to convert a resource definition file into an executable application. You rarely change anything but your CODE resources during development, and the resource compiler spends a lot of time compiling other resources which have not changed since they were originally created.

To save time, some developers have adopted the technique of storing all of these "static" resources in a separate resource file. This file should be placed on the same volume as your application; when your application starts up, use `OpenResFile` to open the separate file. This will cause the resource map for the separate file to be searched before the normal application resource file's map (which now contains mostly CODE resources, along with any brand-new resources still being tested).

This will have little or no effect on the rest of your program. Any time that a resource is needed, both resource files will be searched automatically so you **don't** need to change each `GetResource` call. (Actually, having the extra resource file open has a minor impact on memory management, and uses one more file-control block; unless you're using a lot of open files at once, or are running at the limits of available memory without segmentation, this shouldn't affect you.)

Once your application is close to being finished, you can use ResEdit to move all the resources back into the main application file, and remove the extra `OpenResFile` at the beginning of your application. You should do this for any major release (alpha, beta, and any other 'heavy-testing' releases). Other minor modifications (such as fine-tuning dialog box item positions) may also be done with ResEdit at this time.

The only catch is that you must be careful if your application adds resources to its own resource file. Most applications do not do this (it's not really a great idea, and causes problems with file servers).

#47: Customizing Standard File

See also: The Standard File Package

Written by: Jim Friedlander October 11, 1985
Updated: March 1, 1988

This note contains an example program that demonstrates how `SFPGetFile` can be customized using the dialog hook and file filter functions.

`SFPGetFile`'s dialog hook function and file filter function enable you to customize `SFPGetFile`'s behavior to fit the needs of your application. This technical note consists primarily of a short example program that

1) changes the title of the Open button to '`MyOpen`',
2) adds two radio buttons so that the user can choose to display either text files or text files and applications.
3) adds a quit button to the `SFPGetFile` dialog,

All this is done in a way so as to provide compatibility with the Macintosh File System (MFS), the Hierarchical File System (HFS) and (hopefully) future systems. If you have any questions as you read, the complete source of the demo program and the resource compiler input file is provided at the end of this technical note.

Basically, we need to do three things: add our extra controls to the resource compiler input file, write a dialog hook function, and write a file filter function.

Modifying the Resource Compiler Input File

First we need to define a dialog in our resource file. It will be DLOG #128:

```
CONST myDLOGID = 128;
```

and it's Rez description is:

```
resource 'DLOG' (128, purgeable) {
    {0, 0, 200, 349},
    dBoxProc, invisible, noGoAway,
    0x0,
    128,
    "MyGF"
};
```

The above coordinates (0 0 200 349) are from the standard Standard File dialog. If you need to change the size of the dialog to accommodate new controls, change these coordinates. Next we need to add a DITL in our resource file that is the same as the standard HFS DITL #–4000 except for one item. We need to change the left coordinate of UserItem #4, or part of the dialog will be hidden if we're running under MFS:

```
/* [4] */
/* left coordinate changed from 232 to 252 so program will
   work on MFS */
{39, 252, 59, 347},
UserItem {
     disabled
};
```

None of the other items of the DITL should be changed, so that your program will remain as compatible as possible with different versions of Standard File. Finally, we need to add three items to this DITL, two radio buttons and one button (to serve as a quit button)

```
/* [11] textButton */
{1, 14, 20, 142},
RadioButton {
    enabled,
    "Text files only"
};
/* [12] textAppButton */
{19, 14, 38, 176},
RadioButton {
    enabled,
    "Text and applications"
};
/* [13] quitButton */
{6, 256, 24, 336},
Button {
    enabled,
    "Quit"
}
```

Because we've added three items, we need also need to change the item count for the DITL from 10 to 13. We also include the following in our resource file:

```
resource 'STR#' (256) {
     {/* array StringArray: 1 elements */
         /* [1] */
         "MyOpen"
     }
};
```

That's all there is to modify in the resource file.

The Dialog Hook

We will be calling SFPGetFile as follows:

```
SFPGetFile (wher, '', @SFFileFilter, NumFileTypes,
            MyFileTypes, @MySFHook, reply, myDLOGID,nil);
```

Notice that we're passing @MySFHook to Standard File. This is the address of our dialog hook routine. Our dialog hook is declared as:

```
FUNCTION MySFHook(MySFitem: INTEGER; theDialog: DialogPtr):INTEGER;
```

A dialog hook routine allows us to see every item hit before standard file acts on it. This allows us to handle controls that aren't in the standard SFPGetFile's DITL or to handle standard controls in non-standard ways. The dialog hook in this example consists of a case statement with MySFitem as the case selector. Before SFPGetFile displays its dialog, it calls our dialog hook, passing it a –1 as MySFitem. This gives us a chance to initialize our controls. Here we will set the textAppButton to off and the textButton to on:

```
GetDItem(theDialog,textAppButton,itemType,itemToChange,itemBox);
SetCtlValue(controlHandle(itemToChange),btnOff);
GetDItem(theDialog,textButton,itemType,itemToChange,itemBox);
SetCtlValue(controlHandle(itemToChange),btnOn);
```

and we can also change the title of an existing control. Here's how we might change the title of the Open button using a string that we get from a resource file:

```
GetIndString(buttonTitle,256,1);
If buttonTitle <> '' then Begin          { if we really got the resource}
    GetDItem(theDialog,getOpen,itemType,itemToChange,itemBox);
    SetCtitle(controlHandle(itemToChange),buttonTitle);
End; {if}          {if we didn't get the resource, don't change the title }
```

Upon completion of our routine that handles the –1, we return a –1 to standard file:

```
MySFHook:= MySFItem;                    {pass back the same item we were sent}
```

We now have a SFPGetFile dialog displayed that has a quit button and two radio buttons (the textOnly button is on, the TextApp button is off). In addition, the standard Open button has been renamed to MyOpen (or whatever STR is the first string in STR# 256). This was all done before SFPGetFile displayed the dialog. Once our hook is exited, SFPGetFile displays the dialog and calls ModalDialog.

When the user clicks on an item in the dialog, our hook is called again. We can then take appropriate actions, such as highlighting the `textButton` and un-highlighting the `textAppButton` if the user clicks on the `textButton`. At this time, we can also update a global variable (`textOnly`) that we will use in our file filter function to tell us which files to display. Notice that we can redisplay the file list by returning a 101 as the result of `MySFHook`. (Standard File for Systems newer than 4.3 will also read the low memory globals, CurDirStore and SFSaveDisk, and switch directories when necessary if a 101 is returned as the result. Thus, you can point Standard File to a new directory, or a new disk.) For example, when the `textButton` is hit we turn the `textAppButton` off, turn the `textButton` on, update the global variable `textOnly`, and tell `SFPGetFile` to redisplay the list of files the user can choose from:

```
if not textOnly then Begin    {if textOnly was turned off, turn it on now}
    GetDItem(theDialog,textAppButton,itemType,itemToChange,itemBox);
    SetCtlValue(controlHandle(itemToChange),btnOff);
    GetDItem(theDialog,textButton,itemType,itemToChange,itemBox);
    SetCtlValue(controlHandle(itemToChange),btnOn);
    textOnly:=TRUE;     {toggle our global variable for use in the filter}
    MySFHook:= reDrawList;{101}       {we must tell SF to redraw the list}
End;  {if not textOnly}
```

If our quit button is hit, we can pass `SFPGetFile` back the cancel button:

```
MySFHook:= getCancel;
```

If one of `SFPGetFile`'s standard items is hit, it is very important to pass that item back to `SFPGetFile`:

```
MySFHook:= MySFItem; {pass back the same item we were sent}
```

The File Filter

Remember, we called `SFPGetFile` as follows:

```
SFPGetFile (wher, '', @SFFileFilter, NumFileTypes,
            MyFileTypes, @MySFHook, reply,myDLOGID,nil);
```

Notice that we're passing `@SFFileFilter` to `SFPGetFile`. This is the address of our file filter routine. A file filter is declared as:

```
FUNCTION SFFileFilter (p: ParmBlkPtr): BOOLEAN;
```

A file filter routine allows us to control which files `SFPGetFile` will display for the user. Our file filter is called for every file (of the type(s) specified in the typelist) on an MFS disk, or for every file (of the type(s) specified in the typelist) in the current directory on an HFS disk. In addition, `SFPGetFile` displays HFS folders for us automatically. Our file filter selects which files should appear in the dialog by returning FALSE for every file that should be shown and TRUE for every file that shouldn't.

For example, using our global variable `textOnly` (which we set in our dialog hook, remember?):

```
FUNCTION SFFileFilter(p:parmBlkPtr):boolean;

Begin {SFFileFilter}
  SFFileFilter:= TRUE;                              {Don't show it -- default}

  if textOnly then
      if p^.ioFlFndrInfo.fdType = 'TEXT' then
          SFFileFilter:= FALSE                          {Show TEXT files only}
      else Begin
      End  {dummy else}
  else
      if (p^.ioFlFndrInfo.fdType = 'TEXT') or
                (p^.ioFlFndrInfo.fdType = 'APPL') then
                     SFFileFilter:= FALSE;        { show TEXT or APPL files}
End;  {SFFileFilter}
```

`SFPGetFile` calls the file filter after it has called our dialog hook. Please remember that the filter is passed every file of the types specified in the typelist (`MyFileTypes`). If you want your application to be able to choose from all files, pass `SFPGetFile` a –1 as `numTypes`. For information about parameters to `SFPGetFile` that haven't been discussed in this technical note, see the Standard File Package chapter of *Inside Macintosh*.

That's all there is to it!! Now that you know how to modify `SFPGetFile` to suit your needs, please don't rush off and load up the dialog window with all kinds of controls and text. Please make sure that you adhere to Macintosh interface standards. Similar techniques can be used with `SFGetFile`, `SFPutFile` and `SFPPutFile`.

The complete source of the demo program and of the resource compiler input file follows:

MPW Pascal Source

```
{$R-}

{Jim Friedlander    Macintosh Technical Support      9/30/85}

program SFGetDemo;

USES
     MemTypes,
     QuickDraw,
     OSIntf,
     ToolIntf,
     PackIntf;
{$D+}

CONST
  myDLOGID = 128;               {ID of our dialog for use with SFPGetFile}

VAR
  wher: Point;                  { where to display dialog }
  reply: SFReply;               { reply record }
  textOnly: BOOLEAN;            { tells us which files are currently being displayed}
  myFileTypes: SFTypeList;      { we won't actually use this }
  NumFileTypes: integer;


{-------------------------------------------------------------------------------}
FUNCTION MySFHook(MySFitem:integer; theDialog:DialogPtr): integer;

CONST
  textButton                    = 11;     {DITL item number of textButton}
  textAppButton                 = 12;     {DITL item number of textAppButton}
  quitButton                    = 13;     {DITL item number of quitButton}


  stayInSF                      =  0;     {if we want to stay in SF after getting an Open hit,
                                           we can pass back a 0 from our hook  (not used in
                                           this example) }
  firstTime                     = -1;     {the first time our hook is called, it is passed a
                                           -1}

{The following line is the key to the whole routine -- the magic 101!!}
  reDrawList                    = 101;    {returning 101 as item number will cause the
                                           file list to be recalculated}
  btnOn                         = 1;      {control value for on}
  btnOff                        = 0;      {control value for off}

VAR
  itemToChange: Handle;                   {needed for GetDItem and SetCtlValue}
  itemBox:Rect;                           {needed for GetDItem}
  itemType:integer;                       {needed for GetDItem}
  buttonTitle: Str255;                    {needed for GetIndString}

Begin {MySFHook}
  case MySFItem of

    firstTime: Begin                      { before the dialog is drawn, our hook gets called
                                            with a -1 (firstTime) as the item so we can change
                                            things like button titles, etc. }
```

```
{Here we will set the textAppButton to OFF, the textButton to ON}
        GetDItem(theDialog,textAppButton,itemType,itemToChange,itemBox);
        SetCtlValue(controlHandle(itemToChange),btnOff);
        GetDItem(theDialog,textButton,itemType,itemToChange,itemBox);
        SetCtlValue(controlHandle(itemToChange),btnOn);

        GetIndString(buttonTitle,256,1);
                                        {get the button title from a resource file}
        If buttonTitle <> '' then Begin     { if we really got the resource}
            GetDItem(theDialog,getOpen,itemType,itemToChange,itemBox); {get a handle to the
                                                                        open button}
            SetCtitle(controlHandle(itemToChange),buttonTitle);
        End; {if}                       {if we can't get the resource, we just won't change
                                          the open button's title}
        MySFHook:= MySFItem;            {pass back the same item we were sent}
    End;  {firstTime}


{Here we will turn the textAppButton OFF, the textButton ON and redraw the list}
    textButton: Begin
        if not textOnly then Begin
            GetDItem(theDialog,textAppButton,itemType,itemToChange,itemBox);
            SetCtlValue(controlHandle(itemToChange),btnOff);
            GetDItem(theDialog,textButton,itemType,itemToChange,itemBox);
            SetCtlValue(controlHandle(itemToChange),btnOn);
            textOnly:=TRUE;
            MySFHook:= reDrawList;      {we must tell SF to redraw the list}
        End;  {if not textOnly}
    End;  {textOnlyButton}

{Here we will turn the textButton OFF, the textAppButton ON and redraw the list}
    textAppButton: Begin
        if textOnly then Begin
            GetDItem(theDialog,TextButton,itemType,itemToChange,itemBox);
            SetCtlValue(controlHandle(itemToChange),BtnOff);
            GetDItem(theDialog,TextAppButton,itemType,itemToChange,itemBox);
            SetCtlValue(controlHandle(itemToChange),BtnOn);
            TextOnly:=FALSE;
            MySFHook:= reDrawList;      {we must tell SF to redraw the list}
          End;  {if not textOnly}
        End;  {textAppButton}

    quitButton:  MySFHook:= getCancel;  {Pass SF back a 'cancel button'}

{!!!!very important !!!! We must pass SF's 'standard' item hits back to SF}
    otherwise  Begin
                MySFHook:= MySFItem;    { the item hit was one of SF's standard items... }
    End;  {otherwise}                   { so just pass it back}
  End;  {case}
End;  {MySFHook}

{-------------------------------------------------------------------------------}
```

```
FUNCTION SFFileFilter(p:parmBlkPtr):boolean; {general strategy -- check value of global var
                                              textOnly to see which files to display}

Begin {SFFileFilter}
  SFFileFilter:= TRUE;                      {Don't show it -- default}

  if textOnly then
      if p^.ioFlFndrInfo.fdType = 'TEXT' then
          SFFileFilter:= FALSE              {Show it}
      else Begin
      End  {dummy else}
  else
      if (p^.ioFlFndrInfo.fdType = 'TEXT') or (p^.ioFlFndrInfo.fdType = 'APPL') then
          SFFileFilter:= FALSE;             {Show it}
End;  {SFFileFilter}


{-------------------------------------------------------------------------------}

Begin {main program}
   InitGraf (@thePort);
   InitFonts;
   InitWindows;
   TEInit;
   InitDialogs (nil);

   wher.h:=80;
   wher.v:=90;
   NumFileTypes:= -1;                       {Display all files}

{ we don't need to initialize MyFileTypes, because we want to get a chance to filter every file
  on the disk in SFFileFilter - we will decide what to show and what not to. If you want to
  filter just certain types of files by name, you would set up MyFileTypes and NumFileTypes
  accordingly}

   repeat
       textOnly:= TRUE;{each time SFPGetFile is called, initial display will be text-only
                                files}
       SFPGetFile (wher, '', @SFFileFilter, NumFileTypes, MyFileTypes, @MySFHook,
                                reply,myDLOGID,nil);
   until reply.good = FALSE;
          {until we get a cancel button hit ( or a Quit button -- thanks to our dialog hook ) }
End.
```

MPW C Source

```
#include <Types.h>
#include <Quickdraw.h>
#include <Resources.h>
#include <Fonts.h>
#include <Windows.h>
#include <Menus.h>
#include <TextEdit.h>
#include <Events.h>
#include <Dialogs.h>
#include <Packages.h>
#include <Files.h>
#include <Controls.h>
#include <ToolUtils.h>
```

```
     /*DITL item number of textButton*/
#define      textButton            11

     /*DITL item number of textAppButton*/
#define      textAppButton 12

     /*DITL item number of quitButton*/
#define      quitButton            13

     /*if we want to stay in SF after getting an Open hit, we can pass back a 0
    from our hook  (not used in this example) */
#define      stayInSF              0

     /*the first time our hook is called, it is passed a -1*/
#define      firstTime      -1

     /*The following line is the key to the whole routine -- the magic 101!!*/
     /*returning 101 as item number will cause the file list to be recalculated*/
#define      reDrawList     101

     /*control value for on*/
#define      btnOn          1

     /*control value for off*/
#define      btnOff         0

     /*resource ID of our DLOG for SFPGetFile*/
#define myDLOGID    128

Boolean             textOnly;             /* tells us which files are currently being
displayed*/

main()
{    /*main program*/

            pascal short MySFHook();
            pascal Boolean flFilter();

            Point          wher;                 /* where to display dialog */
            SFReply        reply;
     /* reply record */
            SFTypeList     myFileTypes;
     /* we won't actually use this */
            short int      NumFileTypes = -1;

    InitGraf(&qd.thePort);
    InitFonts();
    FlushEvents(everyEvent, 0);
    InitWindows();
    TEInit();
    InitDialogs(nil);
    InitCursor();


   wher.h=80;
   wher.v=90;
```

```
/* we don't need to initialize MyFileTypes, because we want to get a chance to filter every
file on  the disk in flFilter - we will decide what to show and what not to. if you want to
filter just certain types of files by name, you would set up MyFileTypes and NumFileTypes
accordingly*/

   do
   {    textOnly= true;     /*each time SFPGetFile is called, initial display will be
     text-only files*/
       SFPGetFile(&wher, "",flFilter, NumFileTypes,myFileTypes,MySFHook, &reply,myDLOGID,nil);
   }while (reply.good);    /*until we get a cancel button hit ( or a Quit button in this case )
*/
} /* main */


pascal short MySFHook(MySFItem,theDialog)
short MySFItem;
DialogPtr theDialog;

{

Handle  itemToChange;              /*needed for GetDItem and SetCtlValue*/
Rect  itemBox;                     /*needed for GetDItem*/
short itemType;                   /*needed for GetDItem*/
char  buttonTitle[256];           /*needed for GetIndString*/

  switch (MySFItem)
  {
     case firstTime:
            /* before the dialog is drawn, our hook gets called with a -1 (firstTime)...*/
            /* as the item so we can change things like button titles, etc. */
            /*Here we will set the textAppButton to OFF, the textButton to ON*/
            GetDItem(theDialog,textAppButton,&itemType,&itemToChange,&itemBox);
            SetCtlValue(itemToChange,btnOff);
            GetDItem(theDialog,textButton,&itemType,&itemToChange,&itemBox);
            SetCtlValue(itemToChange,btnOn);

            GetIndString((char *)buttonTitle,256,1);
                   /*get the button title from a resource file*/
            if (buttonTitle[0] != 0)     /* check the length of the p-string to
                                          see if we really got the resource*/
            {
                   GetDItem(theDialog,getOpen,&itemType,&itemToChange,&itemBox); /*get a
                                         handle to the open button*/
                   SetCTitle(itemToChange,buttonTitle);
            }  /*if we can't get the resource, we just won't change the open button's title*/
            return MySFItem;       /*pass back the same item we were sent*/
            break;

/*Here we will turn the textAppButton OFF, the textButton ON and redraw the list*/
    case textButton:
        if (!textOnly)
            {
                   GetDItem(theDialog,textAppButton,&itemType,&itemToChange,&itemBox);
                   SetCtlValue(itemToChange,btnOff);
                   GetDItem(theDialog,textButton,&itemType,&itemToChange,&itemBox);
                   SetCtlValue(itemToChange,btnOn);
                   textOnly=true;
                   return(reDrawList);
                  /*we must tell SF to redraw the list*/
            }  /*if !textOnly*/
            return MySFItem;
            break;
```

```
/*Here we will turn the textButton OFF, the textAppButton ON and redraw the list*/
    case textAppButton:
        if (textOnly)
             {
                   GetDItem(theDialog,textButton,&itemType,&itemToChange,&itemBox);
                   SetCtlValue(itemToChange,btnOff);
                   GetDItem(theDialog,textAppButton,&itemType,&itemToChange,&itemBox);
                   SetCtlValue(itemToChange,btnOn);
                   textOnly=false;
                   return(reDrawList);
                  /*we must tell SF to redraw the list*/
            }  /*if not textOnly*/
            return MySFItem;       /*pass back the same item we were sent*/
            break;

    case quitButton:
            return(getCancel);
           /*Pass SF back a 'cancel button'*/

/*!!!!!!very important !!!!!!!! We must pass SF's 'standard' item hits back to SF*/
    default:
            return(MySFItem);     /* the item hit was one of SF's standard items... */
  }  /*switch*/
     return(MySFItem);     /* return what we got */
}  /*MySFHook*/


pascal Boolean flFilter(pb)
FileParam    *pb;

{

/* is this gross or what??? */
return((textOnly) ? ((pb->ioFlFndrInfo.fdType) != 'TEXT') :
     ((pb->ioFlFndrInfo.fdType) != 'TEXT') &&
     ((pb->ioFlFndrInfo.fdType) != 'APPL'));
}  /*flFilter*/
```

Rez Input File

```
#include "types.r"

resource 'STR#' (256) {
  {
    "MyOpen"
  }
};

resource 'DLOG' (128, purgeable) {
  {0, 0, 200, 349},
  dBoxProc,
  invisible,
  noGoAway,
  0x0,
  128,
  "MyGF"
};
```

```
resource 'DITL' (128, purgeable) {
  {
    /* [1] */
    {138, 256, 156, 336},
    Button { enabled, "Open" };
    /* [2] */
    {1152, 59, 1232, 77},
    Button { enabled, "Hidden" };
    /* [3] */
    {163, 256, 181, 336},
    Button { enabled, "Cancel" };
    /* [4] */
    {39, 252, 59, 347},
    UserItem { disabled };
    /* [5] */
    {68, 256, 86, 336},
    Button { enabled, "Eject" };
    /* [6] */
    {93, 256, 111, 336},
    Button { enabled, "Drive" };
    /* [7] */
    {39, 12, 185, 230},
    UserItem { enabled };
    /* [8] */
    {39, 229, 185, 245},
    UserItem { enabled };
    /* [9] */
    {124, 252, 125, 340},
    UserItem { disabled };
    /* [10] */
    {1044, 20, 1145, 116},
    StaticText { disabled, "" };
    /* [11] */
    {1, 14, 20, 142},
    RadioButton { enabled, "Text files only" };
    /* [12] */
    {19, 14, 38, 176},
    RadioButton { enabled, "Text and applications" };
    /* [13] */
    {6, 256, 24, 336},
    Button { enabled, "Quit" }
  }
};
```

#48: Bundles

See also: The Finder Interface

Written by: Ginger Jernigan November 1, 1985
Updated: March 1, 1988

This note describes what a bundle is and how to create one.

A bundle is a collection of resources. Bundles can be used for a number of different purposes, and are currently used by the Finder ito tie an icon to a file type, allowing your application or data file to have its own icon.

How to Create a Bundle

A bundle is a collection of resources. To make a bundle for finder icons, we need to set up four types of resources: an ICN#, an FREF, a creator STR and a BNDL.

The ICN# resource type is an icon list. Each ICN# resource contains one or more icons, on after another. For Finder bundle icons, there are **two** icons in each ICN#: one for the icon itself and one for the mask. In our sample bundle, we have two file types, each with its own icon. To define the icons for these files we would enter this into our Rez input file:

```
resource 'ICN#' (732) { /* first icon: the ID number can be anything */
   {                    /* first, the icon */
      $"FF FF FF FF"    /* each line is 4 bytes (32 bits) */
      $"F0 09 CD DD"    /* 32 lines total for icon */
      ...
      $"FF FF FF FF"    /* 32nd line of icon */
      ,                 /* now, the mask */
      $"FF FF FF FF"    /* 32 lines total for mask */
      $"FF FF FF FF"
      ...
      $"FF FF FF FF"    /* 32nd line of mask*/
   }
};
resource 'ICN#' (733) { /* second icon */
   {
      $"FF FF FF FF"
      ...
      ,
      $"FF FF FF FF"
      ...
   }
};
```

Now that we've defined our icons we can set up the FREFs. An FREF is a file type reference; you need one for each file type that has an icon. It ties a file type to a local icon resource ID. This will be mapped by the BNDL onto an actual resource ID number of an ICN# resource. Our FREFs will look like this:

```
resource 'FREF' (816) { /* file type reference for application icon */
   {
      'APPL', 605, /* the type is APPL(ication), the local ID is 605 */
      ""           /* this string should be empty (it is unused) */
   }
};

resource 'FREF' (816) { /* file type reference for a document icon */
   {
      'TEXT', 612, /* the type is TEXT, the local ID is 612 */
      ""           /* this string should be empty (it is unused) */
   }
};
```

The reason that you specify the local ID, rather than the actual resource ID of the ICN# is that the Finder will copy all of the bundle resources into the Desktop file and renumber them to avoid conflicts. This means that the actual IDs will change, but the local IDs will remain the same.

Every application (or other file with a bundle) has a unique four-character signature. The Finder uses this to identify an application. The creator resource that contains a single string, and should be defined like this:

```
type 'MINE' as 'STR '; /* MINE is the signature */
resource 'MINE' (0) { /* the creator resource ID must be 0 */
   "MyProgram 1.0 Copyright 1988"
};
```

Now for the BNDL resource. The BNDL resource associates local resource IDs with actual resource IDs, and also tells the Finder what file types exist, and which ICN#s and FREFs are part of the bundle. The resource looks like this:

```
resource 'BNDL' (128) { /* the bundle resource ID should be 0 */
   'MINE', /* signature of this application */
   0, /* the creator resource ID (this must be 0) */
   {
      'ICN#', /* local resource ID mapping for icons */
      {
         605, 732, /* ICN# local ID 605 maps to 732 */
         612, 733  /* ICN# local ID 612 maps to 733 */
      },
      'FREF', /* local resource ID mapping for file type references */
      {
         523, 816, /* FREF local ID 523 maps to 816 */
         555, 817  /* FREF local ID 555 maps to 817 */
      },
```

When you are in the Finder, your application, type APPL (FREF 816), will be displayed with icon local ID 605 (from the FREF resource). This is ICN# 732. Files of type TEXT (FREF 817) created by your application will be displayed with icon local ID 612 (from the FREF resource). This is ICN# 733.

How the Finder Uses Bundles

If a file has the bundle bit set, but the bundle isn't in the Desktop file, the Finder looks for a BNDL resource. If the BNDL resource matches the signature of theapplication, the Finder then makes a copy of the bundle and puts it in the Desktop file. The file is then displayed with its associated icon.

If a file has lost its icon (it's on a disk without the file containing bundle and the Desktop file doesn't contain the bundle), then it will be displayed with the default document icon until the Finder encounters a copy of the file that contains the right bundle. The Finder then makes a copy of the application's bundle (renumbering resources if necessary) and places it in the Desktop file of that disk.

Problems That May Arise

There are times when you have set up these resource types properly but the icon is either the wrong one or it has defaulted to the standard application or data file icon. There are a number of possible reasons for this.

If you are using the Macintosh-based RMaker, the first thing to check is whether there are any extraneous spaces in your resource compiler input file. The Macintosh-based RMaker is very picky about extra spaces.

If your icon is defaulting to the standard icon, check to see that the bundle bit is set. If the bundle bit isn't set , the Finder doesn't know to place the bundle in the Desktop file. If it isn't in the Desktop file, the Finder displays the file with a default icon.

If you changed the icon and remade the resource file, but the file still has the same old icon when displayed in the Finder. The old icon is still in the Desktop file. The Finder doesn't know that you've changed it, so it uses what it has. To get it to use the new icon you need to rebuild the Desktop file. To force the Finder to rebuild the Desktop file, you can hold down the Option and Command keys on startup or on insertion of the disk in question if it isn't the boot disk. The Finder will ask whether or not you want to rebuild the desktop (meaning the Desktop file).

Have a bundle of fun!

#50: Calling SetResLoad

See also: The Resource Manager
Technical Note #1—DAs and System Resources

Written by: Jim Friedlander October 25, 1985
Updated: March 1, 1988

Calling `SetResLoad(FALSE)` can be useful if you need to get a handle to a resource, without causing the resource to be loaded from disk if it isn't already in memory. This technique is used in Technical Note #1. `SetResLoad` changes the value of the low-memory global `ResLoad` (at location `$A5E`).

It is very important that your program not leave `ResLoad` set to `FALSE` when it exits. Doing this will cause the system to reboot or crash when it does a `GetResource` call for the next code segment to be loaded (usually the Finder). The system will crash because `GetResource` will not actually load the code from disk when `ResLoad` is `FALSE`.

So, make sure that you call `SetResLoad(TRUE)` before exiting your program.

#51: Debugging With PurgeMem and CompactMem

See also: The Memory Manager

Written by: Jim Friedlander October 19, 1985
Updated: March 1, 1988

If you are having problems finding bugs like handles that aren't locked down when they should be, or resources that aren't there when they're supposed to be, there is a handy technique for forcing these problems to the surface. Every time through the main event loop call:

```
PurgeMem(MaxSize);                          {MaxSize = $800000}
size:= CompactMem(MaxSize);
```

`PurgeMem` will purge all purgeable blocks and `CompactMem` will rearrange the heap, trying to find a contiguous free block of `MaxSize` bytes. Obviously, this will move things around quite a bit, so, if there are any unlocked handles that you have de-referenced, you will find out about them very quickly.

Don't be alarmed when you see the performance of your program deteriorate drastically —it's because lots of resources are being loaded and purged every time through the main event loop. You might want to have a debugging menu item that toggles between glacial and normal execution speeds.

Please be sure to remove these two lines from any code that you ship!! In fact, neither of these two calls should normally be made from your application. They tend to undo work that has been done by the Memory and Resource Managers.

Macintosh
Technical Notes



Developer Technical Support

#52: Calling _Launch From a High-Level Language

Revised by: Rich Collyer
Written by: Jim Friedlander

April 1989
November 1985

This Technical Note formerly discussed calling `_Launch` from a high-level language which allows inline assembly code.
Changes since March 1988: Merged contents into Technical Note #126.

This Note formerly discussed calling `_Launch` from a high-level language. The information on calling `_Launch` is now contained in Technical Note #126, Sub(Launching) From a High-Level Language, which also covers sublaunching other applications.

#53: MoreMasters Revisited

See also: The Memory Manager

Written by: Jim Friedlander
Updated:

October 28, 1985
March 1, 1988

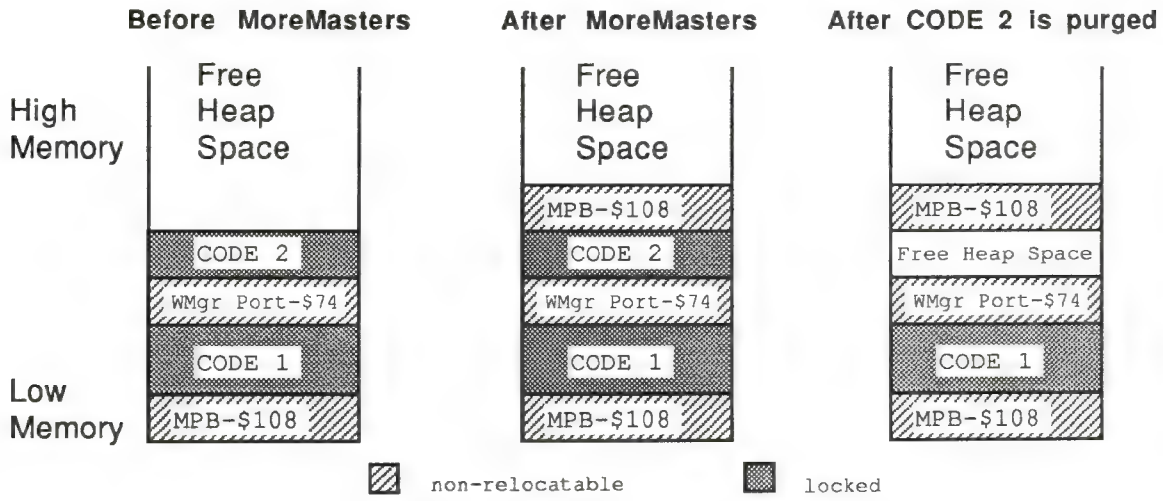
`MoreMasters` should be called from CODE segment 1. The number of master pointers that a program needs can be determined empirically. `MoreMasters` can be tricked into creating the exact number of master pointers desired.

If you ask Macintosh programmers when and how many times `MoreMasters` should be called, you will get a variety of answers, ranging from "four times in the initialization segment" to "once, anywhere." As you might suspect, the answer is somewhat different from either of these.

`MoreMasters` allocates a block of master pointers in the current heap zone. In the application heap, a block of master pointers consists of 64 master pointers; in the system heap, a block consists of 32 master pointers. Since master pointer blocks are **non-relocatable**, we want to be sure to allocate them early. The system will allocate one master pointer block as your program loads. It's the first object in the application heap—its size is $108 bytes.

A lot of programmers call `MoreMasters` from an "initialization" segment, but as we shall see, that's not such a good idea. The problem occurs when we unload our "initialization" segment and it gets purged from memory.

The following diagrams of the application heap illustrate what happens if we call `MoreMasters` from CODE segment 2 (MPB stands for Master Pointer Block):



Notice that we now have some heap fragmentation—not serious, but it can be avoided by making all `MoreMasters` calls in CODE segment 1. Because `InitWindows` creates the Window Manager Port (`WMgrPort`), it should also be called from CODE segment 1. Both `MoreMasters` and `InitWindows` should be called before another CODE segment is loaded, or the non-relocatable objects they allocate will be put above the CODE segment and you'll get fragmentation when the CODE segment is purged. If you want to call an initialization segment before calling `MoreMasters` and `InitWindows`, make sure that you unload it before you call either routine.

Now that we know when to call `MoreMasters`, how many times do we call it? The answer depends on your application. If you don't call `MoreMasters` enough times, the system will call it when it needs more master pointers. This can happen at very inconvenient times, causing heap fragmentation. If you call `MoreMasters` too often, you can be wasting valuable memory. This is preferable, however, to allocating too few master pointer blocks!

The number of times you should call `MoreMasters` can be empirically determined. Once your application is almost finished, remove all `MoreMasters` calls. Exercise your application as completely as possible, opening windows, using handles, opening desk accessories, etc. You can then go in with a debugger and see how many times the system called `MoreMasters`. You do that by counting the non-relocatables of size $108. Due to Memory Manager size correction, the master pointer blocks can also have a size of $10C or $110 bytes. You should give yourself about 20% leeway — that is, if the system called `MoreMasters` 10 times for you, you should call it 12 times. If you're more cautious, you might want to call `MoreMasters` 15 times.

Another technique that can save time at initialization is to calculate the number of master pointers you will need, then set the `MoreMast` files of the heap zone header to that number, and then call `MoreMasters` once:

```
PROCEDURE MyMoreMasters(numMastPtrs : INTEGER);

VAR
   oldMoreMast : INTEGER;       {saved value of MoreMast}
   zone        : THz;           {heap zone}

BEGIN
   zone := GetZone;             {get the heap zone}
   WITH zone^ DO BEGIN
      oldMoreMast := MoreMast; {get the old value of MoreMast}
      MoreMast := numMastPtrs; {put the value we want in the zone header}
      MoreMasters;             {allocate the master pointers}
      MoreMast := oldMoreMast; {restore the old value of MoreMast}
   END;
END;
```

In MPW C:

```
void MyMoreMasters(numMastPtrs)
short numMastPtrs;

{ /* MyMoreMasters */
   short     oldMoreMast;          /* saved value of MoreMast*/
   THz       oZone;                /* heap zone*/

   oZone = GetZone();              /* get the heap zone*/
   oldMoreMast = oZone->moreMast;  /* get the old value of MoreMast*/
   oZone->moreMast = numMastPtrs;  /* put the value we want in the
                                      zone header */
   MoreMasters();                  /*allocate the master pointers*/
   oZone->moreMast = oldMoreMast;  /*restore the old value of MoreMast*/
} /* MyMoreMasters */
```

#54: Limit to Size of Resources

Written by: Jim Friedlander
Updated:

October 23, 1985
March 1, 1988

This note formerly described a bug in `WriteResource` on 64K ROM machines. Information specific to 64K ROM machines has been deleted from Macintosh Technical Notes for reasons of clarity.

#55: Drawing Icons

See also: QuickDraw
Toolbox Utilities

Written by: Jim Friedlander
Updated:

October 21, 1985
March 1, 1988

Using resources of type ICON allows drawing of icons in `srcOr` mode. Using resources of type ICN# allows for more variety when drawing icons.

There are two different kinds of resources that contain icons: ICON and ICN#. An ICON is a 32 by 32 bit image of an icon and can be drawn using the following Toolbox Utilities calls:

```
MyIconHndl:= GetIcon(iconID);
PlotIcon(destRect,iconID);
```

While very convenient, this method only allows the drawing of icons in SrcOr mode (as in the MiniFinder). The Finder uses resources of type ICN# to draw icons on the desktop. Because the Finder uses ICN#s, it can draw icons in a variety of ways.

An ICN# resource is a list of 32 by 32 bit images that are grouped together. Common convention has been to group two 32 by 32 bit images together in each ICN#. The first image is the actual icon, the second image is the mask for the icon. To get a handle to an ICN#, we would use something like this:

```
TYPE
  iListHndl    = ^iListPtr;
  iListPtr     = ^iListStruct;
  iListStruct  = record
                   icon : packed array[0..31] of Longint;
                   mask : packed array[0..31] of Longint;
                 End; {iListStruct}

VAR
   myILHndl      : iListHndl;                    {handle to an ICN#}
   iBitMap       : BitMap;                      {BitMap for the icon}
   mBitMap       : BitMap;                      {BitMap for the mask}

MyILHndl:= iListHndl(GetResource('ICN#',iconID));
if MyILHndl = NIL then HandleError; { and exit or whatever is appropriate}
```

Once we have a handle to the icons, we need to set up two bitMaps that we will be using later in `CopyBits`:

```
SetRect(icnRect,0,0,32,32);                    { define the icon's 'bounds'}
With iBitMap do Begin
    baseAddr:= @MyILHndl^^.icon;
    rowbytes:= 4;                                          { 4 * 8 =32}
    bounds:= icnRect;
End; {with}
With mBitMap do Begin
    baseAddr:= @MyILHndl^^.mask;
    rowbytes:= 4;
    bounds:= icnRect;
End; {with}
```

Icons can represent desktop objects that are either selected or not. Folder and volume icons can either be open or not. The object (or the volume it is on) can either be online or offline. The Finder draws icons using all permutations of open, selected and online:

	Non-Open Non-Selected	Non-Open Selected	Open Non-Selected	Open Selected
Online				
Offline				

Drawing icons as non-open is basically the same for online and offline volumes. We need to punch a hole in the desktop for the icon. This is analogous to punching a hole in dough with an irregular shaped cookie-cutter. We can then sprinkle jimmies* all over the cookie and they will only stick in the area that we punched out (the mask). We do this by copyBitsing the mask onto the desktop (whatever pattern) to our destRect. For non-open, non-selected icons:



we use the SrcBic mode so that we punch a white hole:

```
SetRect(destRect,left,top,left+32,top+32);
CopyBits(mBitMap,thePort^.portBits,icnRect,destRect,SrcBic,NIL);
```

Then we XOR in the icon:

```
CopyBits(iBitMap,thePort^.portBits,icnRect,destRect,SrcXor,NIL);
```

That's all there is to drawing an icon as non-open, non-selected. To draw the icon as non-open, selected:



we will OR in the mask, causing a mask-shaped BLACK hole to be punched in the desktop:

```
CopyBits(mBitMap,thePort^.portBits,icnRect,destRect,SrcOr,NIL);
```

Then, as before, we XOR in the icon:

```
CopyBits(iBitMap,thePort^.portBits,icnRect,destRect,SrcXOr,NIL);
```

To draw icons as non-opened for offline volumes:



we need to do a little more work. We need to XOR a ltGray pattern into the boundsRect of the icon. We will then punch the hole, draw the icon and then XOR out the ltgray pattern that does not fall inside the mask. So, to draw the icon as offline, non-open, non-selected we would:

```
GetPenState(OldPen);              {save the pen state so we can restore it}
PenMode(patXor);
PenPat(ltGray);
PaintRect(destRect);                    {paint a ltGray background for icon}

CopyBits(mBitMap,thePort^.portBits,icnRect,destRect,SrcBic,NIL);    {punch}
PaintRect(destRect);  {XOR out bits outside of the mask, leaving the mask}
                                                        {filled with ltGray}
CopyBits(iBitMap,thePort^.portBits,icnRect,destRect,SrcOr,NIL);  { OR in }
                                                  { the icon to the ltGray mask}
SetPenState(OldPen);                                 {restore the old pen state}
```

To draw the icon as offline, non-open, selected:



we would use a similar approach:

```
GetPenState(OldPen);              { save the pen state so we can restore it}
PenMode(patXor);
PenPat(dkGray);                  { the icon is selected, so we need dkGray }
PaintRect(destRect);                 { paint a dkGray background for icon }

CopyBits(mBitMap,thePort^.portBits,icnRect,destRect,SrcBic,NIL);    {punch}
PaintRect(destRect);  {XOR out bits outside of the mask, leaving the mask}
                                                        {filled with dkGray}
CopyBits(iBitMap,thePort^.portBits,icnRect,destRect,SrcBic,NIL); {BIC the}
                                                    {icon to the dkGray mask}
SetPenState(OldPen);                                 {restore the old pen state}
```

Drawing the opened icons requires one less step. We don"t have to `CopyBits` the icon in, we just use the mask. Online and offline icons are drawn the same way. To draw icons as open, selected:



we do the following:

```
GetPenState(OldPen);              {save the pen state so we can restore it}
PenMode(patXor);
PenPat(dkGray);                 { the icon is selected, so we need dkGray }
PaintRect(destRect);                 { paint a dkGray background for icon}
CopyBits(mBitMap,thePort^.portBits,icnRect,destRect,SrcBic,NIL);    {punch}
PaintRect(destRect);  {XOR out bits outside of the mask, leaving the mask}
                                                        {filled with dkGray}
SetPenState(OldPen);                                 {restore the old pen state}
```

To draw icons as open, non-selected:



we just need to change one line from above. Instead of XORing with a dkGray pattern, we use a ltGray pattern:

```
PenPat(ltGray);              { the icon is non-selected, so we need ltGray }
```

These techniques will work on any background, window-white or desktop-gray and all patterns in between. Have fun.

* *jimmies* : little bits of chocolate

#56: Break/CTS Device Driver Event Structure

See also: The Device Manager
Serial Drivers
Zilog Z8030/Z8530 SCC Serial Communications Controller Technical Manual

Written by: Mark Baumwell
Updated:

December 2, 1986
March 1, 1988

This technical note documents the event record information that gets passed when the serial driver posts an event for a break/CTS status change.

The serial driver can be programmed to post a device driver event upon encountering a break status change or CTS change (via the `SerHShake` call). The structure of device driver events is driver-specific. This technical note documents the event record information that gets passed when the serial driver posts a device driver event for a break/CTS status change.

When the event is posted, the message field of the event record will be a long word (four bytes). The most significant byte will contain the value of SCC Read Register 0 (see below for the relevant Read Register 0 values). The next byte will contain the changed (since the last interrupt) bits of the SCC read register 0. The lower two bytes (word) will contain the `DCtlRefNum`.

The values for Read Register 0 are as follows:

- If a break occurred, bit 7 will be set.
- If CTS changed, bit 5 will reflect the state of the CTS pin (0 means the handshake line is asserted and that it is OK to transmit).

We discourage posting these events because interrupts would be disabled for a long time while the event is being posted. However, it is possible to detect a break or read the value of the CTS line in another way. A break condition will **always** terminate a serial driver input request (but not an output request), and the error breakRecd (–90) will be returned. (This constant is defined in the SysEqu file.) You could therefore detect a break by checking the returned error code.

The state of the CTS line can be checked by making a `SerStatus` call and checking the value of the `ctsHold` flag in the `SerStaRec` record. See the Serial Drivers chapter of *Inside Macintosh* for details.

#57: Macintosh Plus Overview

See: *Inside Macintosh Volume IV*

Written by: Scott Knaster
Updated:

January 8, 1986
March 1, 1988

This note was originally meant as interim Macintosh Plus documentation and has been replaced by *Inside Macintosh Volume IV*, which is more complete and more accurate.

#58: International Utilities Bug

Written by: Jim Friedlander
Updated:

January 24, 1986
March 1, 1988

This note formerly described a bug in System 2.0, which is now recommended only for use with 64K ROM machines. Information specific to 64K ROM machines has been deleted from Macintosh Technical Notes for reasons of clarity.

#59: Pictures and Clip Regions

See also: QuickDraw

Written by: Ginger Jernigan
Updated:

January 16, 1986
March 1, 1988

This note describes a problem that affects creation of QuickDraw pictures.

When a `GrafPort` is created, the fields in the `GrafPort` are given default values; one of these is the clip region, which is set to the rectangle (–32767, –32767, 32767, 32767). If you create a picture, then call `DrawPicture` with a destination rectangle that is not the same size as the `picFrame` without ever changing the default clip region, nothing will be drawn.

When the picture frame is compared with the destination rectangle and the picture is scaled, the clip region is scaled too. In the process of scaling, the clip region you end up overflows and becomes empty, and your picture doesn't get drawn. If you call `ClipRect(thePort^.portRect)` before you record the picture, the picture will be drawn correctly. The clipping on the destination port when playing back the picture is irrelevant: once a picture is incorrectly recorded, it is too late.

#60: Drawing Characters into a Narrow GrafPort

See also: QuickDraw

Written by: Ginger Jernigan
Updated:

January 20, 1986
March 1, 1988

When you draw a character into a `GrafPort`, your program will die with an address error if the width of the `GrafPort` is smaller than the width of the character. If you check before drawing the character to see if the `GrafPort` is wide enough, you can avoid this unfortunate tragedy.

#61: GetItemStyle Bug

Written by: Jim Friedlander
Updated:

January 21, 1986
March 1, 1988

This note formerly described a bug (in `GetItemStyle`) which occurs only on 64K ROM machines. Information specific to 64K ROM machines has been deleted from Macintosh Technical Notes for reasons of clarity.

#62: Don't Use Resource Header Application Bytes

See also: The Resource Manager

Written by: Bryan Stearns
Updated:

January 23, 1986
March 1, 1988

The section of the Resource Manager chapter of *Inside Macintosh* which describes the internal format of a resource file shows an area of the resource header labeled "available for application data." You **should not** use this area—it is used by the Resource Manager.

#63: WriteResource Bug Patch

Written by:	Rick Blair Jim Friedlander Bryan Stearns	January 15, 1986
Modified by :	Jim Friedlander	March 3, 1986
Updated:		March 1, 1988

This note formerly contained a patch to fix a bug in `WriteResource` on 64K ROM machines. Information specific to 64K ROM machines has been deleted from Macintosh Technical Notes for reasons of clarity.

#64: IAZNotify

Written by:	Jim Friedlander	January 15, 1986
Modified by:	Jim Friedlander	August 18, 1986
Updated:		March 1, 1988

Previous versions of this technical note recommended use of a low memory hook called `IAZNotify`. We no longer recommend use of `IAZNotify`, since the `IAZNotify` hook is never called under MultiFinder.

#65: Macintosh Plus Pinouts

See also: *Macintosh Hardware Reference Manual*

Written by: Mark Baumwell January 27, 1986
Modified by: Mark Baumwell March 20, 1986
Updated: March 1, 1988

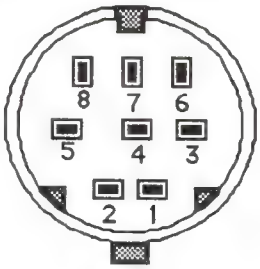
This note gives pinout descriptions for some of the Macintosh Plus ports and Macintosh Plus cables that are different than the Macintosh 128K and 512K.

Below are pinout descriptions for some Macintosh Plus ports and cables that are different than the Macintosh 128K and 512K. Note that any unconnected pins are omitted.

Macintosh Plus Port Pinouts

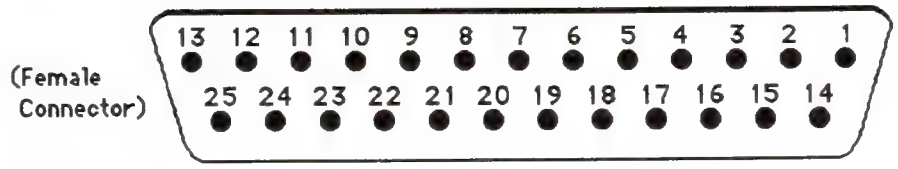
Macintosh Plus Serial Connectors (Mini DIN-8)

(Female Connector)



Pin	Name	Description/Notes
1	HSKo	Output Handshake (from Zilog 8530 DTR pin)
2	HSKi/External Clock	Input Handshake (CTS) or TRxC (depends on 8530 mode)
3	TxD–	Transmit Data line
4	Ground	
5	RxD–	Receive Data line
6	TxD+	Transmit Data line
7	Not connected	
8	RxD+	Receive Data line; ground this line to emulate RS232

Macintosh Plus SCSI Connector (DB-25)

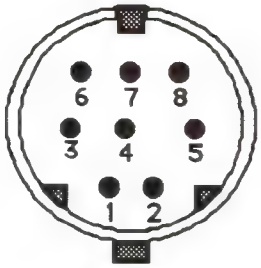


Pin	Name	Description/Notes
1	REQ–	
2	MSG–	
3	I/O–	
4	RST–	
5	ACK–	
6	BSY–	
7	Ground	
8	DB0–	
9	Ground	
10	DB3–	
11	DB5–	
12	DB6–	
13	DB7–	
14	Ground	
15	C/D–	
16	Ground	
17	ATN–	
18	Ground	
19	SEL–	
20	DBP–	
21	DB1–	
22	DB2–	
23	DB4–	
24	Ground	
25	TPWR	Not connected

Macintosh Plus Cable Pinouts

Apple System Peripheral-8 Cable (connects Macintosh Plus to ImageWriter II and Apple Personal Modem)
(Product part number: M0187)
(Cable assembly part number: 590-0340-A (stamped on cable itself).

(Male Connector)



(DIN-8)	(DIN-8)
1	2
2	1
3	5
4	4
5	3
6	8
7	7
8	6

Macintosh Plus Adapter Cable (connects Macintosh Plus DIN-8 to existing Macintosh DB-9 cables)
(Apple part number: M0189)
(Cable assembly part number: 590-0341-A (stamped on cable itself).

(DIN-8)	Name	(DB-9)	Notes
1	+12V	6	
2	HSK	7	
3	TxD–	5	
4	Ground	3	Jumpered to DB-9 pin 1 (in DB-9 connector)
5	RxD–	9	
6	TxD+	4	
7	no wire		
8	RxD+	8	
	Ground	1	Jumpered to DB-9 pin 3 (in DB-9 connector)

Macintosh
Technical Notes



Developer Technical Support

#66: Determining Which File System Is Active

Revised by: Robert Lenoil & Brian Bechtel
Written by: Jim Friedlander

August 1990
December 1985

This Technical Note discusses how to determine which file system a particular volume is running.
Changes since June 1990: Removed text about IDs $0001-$0016 being AppleShare volumes; other file systems use this range too.

Under certain circumstances it is necessary to determine which file system is currently running on a particular volume. For example, on a 64K ROM machine, your application (i.e., especially disk recovery utilities or disk editors, etc.) may need to check for MFS versus HFS. Note that this is usually not necessary, because all ROMs, except the original 64K ROMs, include HFS. If your application only runs on 128K ROMs or newer, you do not need to check for HFS versus MFS. You may need to check if a particular volume is in High Sierra, ISO 9660, or audio CD format.

Before performing these file system checks, be sure to call `_SysEnvirons`, to make sure the machine on which you are running has ROMs which know about the calls you need.

To check for HFS on 64K ROM machines, check the low-memory global `FSFCBLen` (at location `$3F6`). This global is one word in length (two bytes) and is equal to -1 if MFS is active and a positive number (currently `$5E`) if HFS is active. From Pascal, the following would perform the check:

```
CONST
  FSFCBLen = $3F6;    {address of the low-memory global}

VAR
  HFS: ^INTEGER;

  ...
  HFS:= POINTER(FSFCBLen);
  IF HFS^ > 0 THEN
      {we're running HFS}
  ELSE
      {we're running MFS}
  END;
```

If an application determines that it is running under HFS, it should not assume that all mounted volumes are HFS. To check individual volumes for HFS, call `_PBHGetVInfo` and check the directory signature (the `ioVSigWord` field of an `HParamBlockRec`). A directory signature of $D2D7 means the volume is an MFS volume, while a directory signature of $4244 means the volume is an HFS volume.

To find out if a volume uses a file system other than HFS or MFS, call `_PBHGetVInfo` and check the file system ID (the `ioVFSID` field of an `HParamBlockRec`). A file system ID of $0000 means the volume is either HFS or MFS. A file system ID of $4242 means the volume is a High Sierra volume, while a file system ID of $4147 is an ISO 9660 volume, and a file system ID of $4A48 is an audio CD volume. AppleShare and other file systems use a dynamic technique of obtaining the first unused file system ID; therefore, low-numbered IDs cannot be associated with any particular file system.

When dealing with High Sierra and ISO 9660 formats, do not assume that the volumes are CD-ROM discs. Support for these file systems is done with the External File System hook in the File Manager, so any block-based media could potentially be in these formats. It is possible to have a High Sierra formatted floppy disk, although it would be useless except for testing purposes.

Further Reference:

- *Inside Macintosh*, Volume IV, File Manager
- Technical Note #209, High Sierra & ISO 9660 CD-ROM Formats
- Technical Note #129, _SysEnvirons: System 6.0 and Beyond

Macintosh
Technical Notes



Developer Technical Support

#67: How to Bless a Folder to Be the System Folder

Rewritten by: Colleen K. Delgadillo
Updated by: Jim Friedlander
Written by: Jim Friedlander

May 1992
March 1988
January 1986

This Technical Note describes how to determine which folder on an HFS volume is the blessed folder, that is, the folder that contains both the System file and the Finder.

Changes since January 1986: The information about how to find the "Blessed Folder" has been deleted from this technical note. The FindFolder function can now be used to find the "Blessed Folder" and is documented in Inside Macintosh Volume VI, pages 9-42 to 9-44. This note now includes information about how to bless a folder to the new system folder.

Note: The following information may be effected by future changes to system software. If you choose to use this information, you must do so at your own risk.

The way to bless a folder is by taking the longword which is the directory ID of the blessed folder and putting it into the Master Directory Block (MDB). This can be accomplished by using the HFS call `PBSetVInfo`. You should not attempt to change this block directly. First call `PBHGetVInfo` and set `ioVFnderInfo[1]` to the directory ID of the the new folder to be blessed. Then call `PBSetVInfo` to save this information. Once you have done this, you will find that the Finder takes a little while to realize that you have blessed the folder. Therefore, the icon will take a little while to change. Exactly how long you will have to wait to see the new icon is unknown.

Forcing the icon to change sooner is not a difficult task. The best way for you to see the icon change more quickly is to change the modification date of the directory into which you are copying the new System Folder. Doing this will cause the Finder to reexamine the window and its contents. When the Finder notices that the volume's modification date has changed, it begins scanning for changes in all the open folders. This scanning process takes place about once every 10 seconds. You can change the last modification date for that volume and the System Folder's directory ID for that volume using `PBSetVInfo`. Changing the file's `FndrInfo` or renaming the file does not change the modification date. When you call `PBSetVInfo` you will need to put the System Folder's directory ID in the longword at `ioVfndrInfo`. This longword will be the first four bytes of this directory ID. (As usual, whenever you make a change to a field of a structure you need to first do a `PBGetCatInfo`, change what you are going to change, and then do `PBSetCatInfo`. This ensures that you change only the portion of the volume that you intended, in this case a longword, and not the whole structure.)

Further Reference:

- Master Directory Block: Inside Macintosh Volume IV on page 166.

#68: Searching All Directories on an HFS Volume

See also: *Inside Macintosh*, The File Manager
Technical Note #66: Determining Which File System is Active
Technical Note #67: Finding the "Blessed Folder"

Written by: Jim Friedlander December 23, 1985
Updated: March 1, 1988
Revised by: Rick Blair October 1, 1988

This Technical Note provides a simple algorithm for searching all directories on an HFS volume.
Changes since March 1, 1988: Previously we stopped the whole search when we encountered any error, which caused trouble with privilege errors on AppleShare volumes. Now an error will only stop the enumeration of a particular catalog.

Now that we have a Hierarchical File System, it may be necessary to search the hierarchy for files. **WARNING:** This can be a very time consuming process on a large volume. Generally speaking, your application should avoid searching entire volumes, relying instead on files being in specific directories (same directory as the application, or in the "blessed folder") or on having the user find files with Standard File.

Under MFS, indexed calls to `_PBGetFileInfo` would return information about all files on a given volume. Under HFS, the same technique only returns information about files in the current directory.

The most convenient way to search a tree structure is with a recursive search. The only potential problem with a recursive search is that it can use up a lot of stack space. The default stack space on the Macintosh is 8K; therefore, we have to try to keep the stack frame small. This example does that by enclosing the actual recursive routine in a shell that can hold most of the variables that we will need, thus dramatically reducing the size of the stack frame. This example uses only 26 bytes of stack space each time the routine recurses. That is, it could search 100 levels deep (pretty unlikely) and only use 2600 bytes of stack space.

Please notice that when the routine comes back from recursing, it has to clear the non-local variable `err`, since the reason that the routine came back from recursing is that `_PBGetCatInfo` returned an error:

```
EnumerateCatalog(myCPB.ioDrDirID);
        err:= 0; {clear error return on way back}
```

Please notice also that you must set `myCPB.ioDrDirId` each time you call `_PBGetCatInfo`. This is because if `_PBGetCatInfo` gets information about a file, it returns `ioFlNum` (the file number) in the same location that the `ioDrDirID` previously occupied.

One other thing to watch for is when you check the file attributes bit to see if you've got a file or a folder. You need to check bit 4, the fifth least significant bit. Remember, the Toolbox bit manipulation routines order the bits in reverse order from standard 68000 notation. That's why a `BitTst(@myCPB.ioFlAttrib,3)` is used.

Here is the routine in MPW Pascal:

```
PROCEDURE EnumerShell(DirIDToSearch:Longint);
VAR
FName:          Str255;
myCPB:          CInfoPBRec;
err:            OSErr;
myWDPB:         WDPBRec;
TotalFiles,TotalDirectories:        integer;

        PROCEDURE EnumerateCatalog(dirIDToSearch: longint);
        VAR
        index:          integer;

        Begin {EnumerateCatalog}
                index:= 1;
                repeat
                        FName:= '';
                        myCPB.ioFDirIndex:= index;
                        myCPB.ioDrDirID:= dirIDToSearch; {we need to do this every time through}

                        err:= PBGetCatInfo(@myCPB,FALSE);

                        If err = noErr then
                                if BitTst(@myCPB.ioFlAttrib,3) then Begin {we have a dir}
                                        TotalDirectories:=TotalDirectories+1;
                                        EnumerateCatalog(myCPB.ioDrDirID);
                                        err:= 0;  {clear error return on way back}
                                        End {if BitTst}
                                Else Begin {we have a file}
                                        TotalFiles:= TotalFiles + 1;
                                        {here we could call a routine that compares a
                                         string with myCPB.ioNamePtr^}
                                        End; {else}
                        index:= index + 1;
                until (err <> noErr);
        End;  {EnumerateCatalog}

Begin   {EnumerShell}
        TotalFiles:= 0;
        TotalDirectories:= 0;
        myWDPB.ioNamePtr := NIL;
        err:= PBHGetVol(@myWDPB,FALSE);     {get the default volume's vRefNum}

        with MyCPB do Begin
                ioNamePtr:= @FName;
                ioVRefNum:= myWDPB.ioWDVRefNum;     {for now, default vol,
                                                     set this to what you want}
        End; {with}
        EnumerateCatalog(fsRtDirID); {the root level}
End;    {EnumerShell}
```

In MPW C:

```
#include <types.h>
#include <quickdraw.h>
#include <resources.h>
#include <fonts.h>
#include <windows.h>
#include <menus.h>
#include <textedit.h>
#include <events.h>
#include <files.h>
#include <packages.h>
#include <memory.h>


void EnumerateCatalog();                        /* forward declaration */

/* the following variables will be global */

HFileInfo           myCPB;                      /* for the PBGetCatInfo call */
char                fName[256];                 /* place to hold file name */
short int           TotalFiles=0,TotalDirectories=0;

main()

{
        WindowPtr           MyWindow;
        Rect                myWRect;

        InitGraf(&qd.thePort);
        InitFonts();
        FlushEvents(everyEvent, 0);
        InitWindows();
        InitMenus();
        TEInit();

        SetRect(&myWRect,50,260,350,340);
        MyWindow = NewWindow(nil,&myWRect,"\pJim's Window",true,0,(WindowPtr)-1,false,0);
        SetPort(MyWindow);

        /* set up to call EnumerateCatalog */
        myCPB.ioNamePtr = fName;
        myCPB.ioVRefNum = 0;
        /* the default volume */
        EnumerateCatalog(fsRtDirID);            /* the root level */

        /* we're back, now display the total number of files and directories */

        NumToString(TotalFiles,fName);
        MoveTo(20,20);
        EraseRect(&(qd.thePort->portRect));     /* erase what used to be there */
        DrawString("\pTotal Files = ");
        DrawString(fName);

        NumToString(TotalDirectories,fName);
        MoveTo(20,40);
        DrawString("\pTotal Directories = ");
        DrawString(fName);

        MoveTo(20,60);
        DrawString("\pClick To Exit");

        while (!Button());
} /*main*/

/*-------------------------------------------------------------------------*/
```

```
void EnumerateCatalog(dirIDToSearch)
long int      dirIDToSearch;

{ /*EnumerateCatalog*/

      short int     index=1;                    /* for ioFDirIndex */
      OSErr         err;                        /* the usual */

      do
      {
            myCPB.ioFDirIndex= index;           /* set up the index */

/* we need to do this every time through, since GetCatInfo returns ioFlNum in this field*/
            myCPB.ioDirID= dirIDToSearch;
            err= PBGetCatInfo(&myCPB,false);

            if (err == noErr)
            {
                  /* just for demo's sake, we will display the file/folder names */
                  MoveTo(20,20);
                  EraseRect(&(qd.thePort->portRect)); /* clear out the last name */
                  DrawString(fName);                  /* display the name */

                  /* check the file attributes for folderhood */
                  if (((myCPB.ioFlAttrib>>4) & 0x01) == 1)

                  { /*we have a dir*/
                        /* we have a directory, increment the file counter */
                        TotalDirectories += 1;
                        EnumerateCatalog(myCPB.ioDirID);    /* recurse */
                        err = 0;       /* clear error return on way back*/
                  } /*if ((myCPB.ioFlAttrib>>4) & 0x01 == 1)*/
                  else
                  { /*we have a file*/
                        /* we have a file, increment the file counter */
                        TotalFiles += 1;
                        /* we could list the file here */
                  } /*else*/
            index += 1;    /* increment the index for GetCatInfo */
            } /* if (err == noErr) */
      } while (err == noErr);
} /*EnumerateCatalog*/

/* that's all folks!! */
```

Please make sure that you are running under HFS before you use this routine. You can do partial searches of a volume by specifying a starting `dirId` other than `fsRtDirID`.

#69: Setting ioFDirIndex in PBGetCatInfo Calls

See also: The File Manager
Technical Note #24—Available Volumes and Files
Technical Note #67—Finding the Blessed Folder

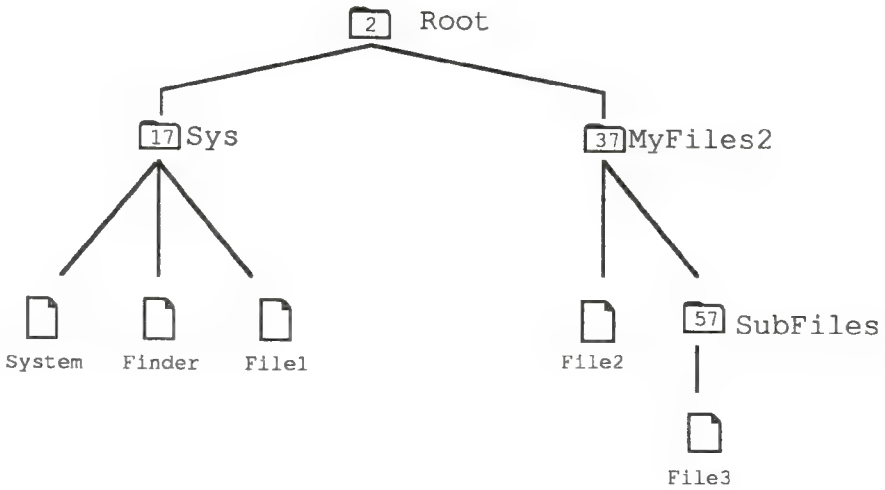
Written by: Jim Friedlander February 15, 1986
Updated: March 1, 1988

This technical note describes how to set `ioFDirIndex` for `PBGetCatInfo`.

The File Manager chapter of *Inside Macintosh* volume IV is not very specific in describing how to use `ioFDirIndex` when calling `PBGetCatInfo`. It correctly says that `ioFDirIndex` should be positive if you are making indexed calls to `PBGetCatInfo` (analogous to making indexed calls to `PBGetVInfo` as described in Technical Note #24). However, the statement "If `ioFDirIndex` is negative or 0, the File Manager returns information about the file having the name in `ioNamePtr`..." is not specific enough.

If `ioFDirIndex` is 0, you will get information about files or directories, depending on what is specified by `ioNamePtr^`.

If `ioFDirIndex` is –1, you will get information about directories only. The name in `ioNamePtr^` is ignored. For example, given the following tree structure (with sample `DirIDs` for the directories):



Calling PBGetCatInfo

We will now make calls to PBGetCatInfo of the form:

```
err:= PBGetCatInfo(@myCInfoPBRec,FALSE);
```

Note: We will assume that we just have a WDRefnum and a file name—the information that SFGetFile returns.

Setting up the parameter block

We will use the following fields in the parameter block. Before the call, ioCompletion will always be set to NIL, ioNamePtr will always point at a str255, ioVRefNum will always contain a WDRefNum that references the directory 'SubFiles', and offset 48 (dirID/flNum) will always contain a zero:

Offset in parameter block	Variable name(s)
12	ioCompletion
18	ioNamePtr
22	ioVRefNum
28	ioFDirIndex
48	ioDirID/ioFLNum/ioDrDirID
100	ioDrParID/ioFlParID

Sample calls to PBGetCatInfo

The first example will call PBGetCatInfo for the file 'File3'—we will get information about the file (ioFDirIndex = 0):

Before the call		After the call	
ioNamePtr^:	'File3'	ioNamePtr^:	'File3'
ioFDirIndex:	**0**	Offset 48(ioFLNum):	a file number
		Offset 100(parID):	57

Now we will get information about the directory that is specified by the iovRefNum (ioFDirIndex = –1). Notice that ioNamePtr^ is ignored:

Before the call		After the call	
ioNamePtr^:	ignored	ioNamePtr^:	'SubFiles'
ioFDirIndex:	**–1**	Offset 48(dirID):	57
		Offset 100(parID):	37

Notice that, since ioNamePtr^ is ignored, Offset 48 contains the dirID of the directory specified by the iovRefNum that we passed in and that Offset 100 contains the parent ID of that directory.
Notice that if we try to get information about the directory SubFiles by calling PBGetCatInfo with ioFDirIndex set to 0, we will get an error –43 (File not found error) back because there is neither a file nor a directory with the name 'SubFiles' in the directory that ioVRefNum refers to.

If you specify a **full** pathname in ioNamePtr^, then the call returns information about that path, whether it is a directory or a file. The ioVRefNum is ignored:

Before the call		**After the call**	
ioNamePtr^:	'Root:Sys'	ioNamePtr^:	'Root:Sys'
ioFDirIndex:	**0**	Offset 48 (dirID):	17
ioVRefNum:	refers to 'SubFiles'	Offset 100 (parID):	2

Or, if the full pathname specifies a file, the iovRefNum is overridden:

Before the call		**After the call**	
ioNamePtr^:	'Root:Sys:Finder'	ioNamePtr^:	'Root:Sys:Finder'
ioFDirIndex:	**0**	Offset 48 (flNum):	fileNumber
ioVRefNum:	refers to 'SubFiles'	Offset 100 (parID):	17

Or, given an ioVRefNum that refers to MyFiles2 and a partial pathname in ioNamePtr^, we'll get information about the directory 'SubFiles':

Before the call		**After the call**	
ioNamePtr^:	'SubFiles'	ioNamePtr^:	'SubFiles'
ioFDirIndex:	**0**	Offset 48 (dirID):	57
ioVRefNum:	refers to 'MyFiles2'	Offset 100 (parID):	37

PBGetCatInfo and The Poor Man's Search Path (PMSP)

If no ioDirID is specified (ioDirID is set to zero), calls to PBGetCatInfo will return information about a file in the specified directory, but, if no such file is found, will continue searching down the Poor Man's Search Path. **Note:** the PMSP is not used if ioFDirIndex is non-zero (either –1 or >0). The default PMSP includes the directory specified by ioVRefNum (or, if ioVRefNum is 0, the default directory) and the directory that contains the System File and the Finder—the blessed folder. So for example:

Before the call		**After the call**	
ioNamePtr^:	'System'	ioNamePtr^:	'System'
ioFDirIndex:	**0**	Offset 48 (ioFLNum):	a file number
		Offset 100 (parID):	17

You must be careful when using PBGetCatInfo in this way to make sure that the file you're getting information about is in the directory that you think it is, and not in a directory further down the Poor Man's Search Path. Of course, this does not present a problem if you are using the fName and the vRefNum that SFGetFile returns.

If you want to specifically look at a file in the blessed folder, please use the technique described in technical note #67 to get the `dirID` of the 'blessed folder' and then use that `dirID` as input in the `ioDirID` field of the parameter block (offset 48).

Summary (`DirID` = 0 in all the following):

If `ioFDirIndex` is set to 0:

1) Information will be returned about files.
2) Information will be returned about directories as follows:
   A) If a partial pathname is specified by `ioNamePtr^` then the volume and directory will be taken from `ioVRefNum`.
   B) If a full pathname is specified by `ioNamePtr^`. In this case, `ioVRefNum` is ignored.

If `ioFDirIndex` is set to –1:

1) Only information about directories will be returned.
2) The name pointed to by `ioNamePtr` is ignored.
3) If `DirID` and `ioVRefNum` are 0, you'll get information about the default directory.

#70: Forcing Disks to be Either 400K or 800K

See also: The Disk Driver
The Disk Initialization Package

Written by: Rick Blair
Updated:

February 13, 1986
March 1, 1988

This document explains how to initialize a disk as either single- or double-sided. It only applies to 800K drives, of course.

You can call the disk driver to initialize a disk and determine programmatically whether it should be initialized as single- (MFS) or double- (HFS) sided. All you have to do is call the `.Sony` driver directly to do the formatting then the Disk Initialization Package to write the directory information.

Note: This is not the way you should normally format disks within an application. If the user puts in an unformatted disk, you should let her or him decide whether it becomes single- or double-sided via the Disk Initialization dialog. This automatically happens when you call DIBadMount or the user inserts a disk while in Standard File. The intent of this technical note is to provide a means for specific applications to produce, say, 400K disks. An example might be a production disk copying program.

From MPW Pascal:

```
VAR
  error:      OSErr;
  IPtr:       ^INTEGER;
  paramBlock: ParamBlockRec; {needs OSIntf}
...
WITH paramBlock DO BEGIN
   ioRefNum := -5;                              {.Sony driver}
   ioVRefNum := 1;                              {drive number}
   csCode := 6;                                 {format control code}
   IPtr:=@csParam;                              {pretend it's an INTEGER}
   IPtr^:=1;                                    {number of sides}
END;
error:=PBControl(@paramBlock, FALSE);           {do the call}
IF error=ControlErr THEN
{you are under MFS, which doesn't support control code 6, but it}
{would always get formatted single-sided anyway.}
{other errors are possible: ioErr, etc.}
END;
```

From MPW C:

```
OSErr             error;
CntrlParam        paramBlock;


paramBlock.ioCRefNum = -5;     /*.Sony driver*/
paramBlock.ioVRefNum = 1;      /*drive number*/
paramBlock.csCode = 6;         /*format control code*/
paramBlock.csParam[0]=1;       /*for single sided,2 for double-sided*/

error=PBControl(&paramBlock, false);/*do the call*/
if (error==controlErr) ;
/*you are under MFS, which doesn't support control code 6, but it*/
/*would always get formatted single-sided anyway.*/
/*other errors are possible: ioErr, etc.*/
```

You then call DIZero to write a standard (MFS or HFS) directory. It will produce MFS if you formatted it single-sided, and HFS if you formatted double-sided.

#71: Finding Drivers in the Unit Table

See also: The Device Manager

Written by: Rick Blair
Updated:

February 4, 1986
March 1, 1988

This note will explain how code can be written to determine the reference number of a previously installed driver when only the name is known. **Changes since 2/86:** Since the driver can be purged and the DCE still be allocated, the code now tests for dCtlDriver being NIL as well.

You should already be familiar with The Device Manager chapter of *Inside Macintosh* before reading this technical note.

The Pascal code at the end of this note demonstrates how to obtain the reference number of a driver that has been installed in the Unit Table. The reference number may then be used in subsequent calls to the Device Manager such as `Open`, `Control` and `Prime`.

One thing to note is that the `dRAMBased` bit really only tells you whether `dCtlDriver` is a pointer or a handle, not necessarily whether the driver is in ROM or RAM. SCSI drivers, for instance, are in RAM but not relocatable; their DCE entries contain pointers to them.

From MPW Pascal:

```
PROCEDURE GetDrvrRefNum(driverName: Str255; VAR drvrRefNum: INTEGER);

   TYPE
      WordPtr      = ^INTEGER;

   CONST
      UTableBase   = $11C;      {low memory globals}
      UnitNtryCnt  = $1D2;

      dRAMBased    = 6;         {bit in dCtlFlags that indicates ROM/RAM}
      drvrName     = $12;       {length byte and name of driver [string]}

   VAR
      negCount     : INTEGER;
      DCEH         : DCtlHandle;
      drivePtr     : Ptr;
      s            : Str255;
```

```
BEGIN
   UprString(driverName, FALSE); {force same case for compare}

   negCount := - WordPtr(UnitNtryCnt)^; {get -(table size)}

   {Check to see that driver is installed, obtain refNum.}
   {Assumes that an Open was done previously -- probably by an INIT.}
   {Driver doesn't have to be open now, though.}

   drvrRefNum := - 12 + 1;  {we'll start with driver refnum = -12,
                             right after .ATP entry}

   {Look through unit table until we find the driver or reach the end.}

   REPEAT
      drvrRefNum := drvrRefNum - 1; {bump to next refnum}
      DCEH := GetDCtlEntry(drvrRefNum); {get handle to DCE}

      s := '';              {no driver, no name}

      IF DCEH <> NIL THEN
         WITH DCEH^^ DO BEGIN {this is safe -- no chance of heap moving
                               before dCtlFlags/dCtlDriver references}
            IF (dCtlDriver <> NIL) THEN BEGIN
              IF BTST(dCtlFlags, dRAMBased) THEN
                 drivePtr := Handle(dCtlDriver)^ {zee deréference}
              ELSE
                 drivePtr := Ptr(dCtlDriver);

              IF drivePtr <> NIL THEN BEGIN
                 s := StringPtr(ORD4(drivePtr) + drvrName)^;
                 UprString(s,FALSE); {force same case for compare}
              END;
            END;              {IF}
         END;                 {WITH}
   UNTIL (s = driverName) OR (drvrRefNum = negCount);

   {Loop until we find it or we've just looked at the last slot.}

   IF s <> driverName THEN drvrRefNum := 0; {can't find driver}
END;
```

From MPW C:

```
short       GetDrvrRefNum(driverName)
char        *driverName[256];

{  /* GetDrvrRefNum */

      #define           UnitNtryCnt 0x1d2

      /*bit in dCtlFlags that indicates ROM/RAM*/
      #define           dRAMBased     6
      /*length byte and name of driver [string]*/
      #define           drvrName      0x12
```

```
        short               negCount,dRef;
        DCtlHandle          DCEH;
        char                *drivePtr,*s;

        negCount = -*(short *)(UnitNtryCnt); /*get -(table size)*/

        /*Check to see that driver is installed, obtain refNum.*/
        /*Assumes that an Open was done previously -- probably by an INIT.*/
        /*Driver doesn't have to be open now, though.*/

        dRef = -12 + 1;  /*we'll start with driver refnum == -12,
                                    right after .ATP entry*/

        /*Look through unit table until we find the driver or reach the
        end.*/

        do
        {
              dRef -= 1; /*bump to next refnum*/
              DCEH = GetDCtlEntry(dRef); /*get handle to DCE*/

              s = "";

              if ((DCEH != nil) && ( (**DCEH).dCtlDriver != nil) )
              {
                    if (((**DCEH).dCtlFlags >> dRAMBased) & 1)
                                                /* test dRamBased bit */
                          drivePtr = *(Handle) (**DCEH).dCtlDriver;
                                                /*zee deréference*/
                    else
                          drivePtr = (**DCEH).dCtlDriver;

                    if (drivePtr != nil)
                          s = drivePtr + drvrName;
              }
        } while (EqualString(s,driverName,0,0) && (dRef != negCount));
        /*Loop until we find it or we've just looked at the last slot.*/

        if (EqualString(s,driverName,0,0))
              return dRef;
        else
              return 0; /*can't find driver*/
  }/* GetDrvrRefNum */
```

That's all there is to locating a driver and picking up the reference number.

Macintosh
Technical Notes



Developer Technical Support

#72: Optimizing For The LaserWriter—Techniques

Revised by: Pete "Luke" Alexander
Written by: Ginger Jernigan

October 1990
February 1986

This Technical Note discusses techniques for optimizing code for printing on the LaserWriter.
Changes since March 1988: Updated the "Printable Paper Area" and "Memory Considerations" sections as well as the printer IDs, moved the error messages from the end of the Note to Technical Note #161, A Printing Loop That Cares..., and removed the "Spool-A-Page/Print-A-Page" section because Technical Note #125, Effect of Spool-A-Page/Print-A-Page on Shared Printers, already thoroughly covers this topic.

Introduction

Although the Printing Manager was originally designed to allow application code to be printer independent, there are some things about the LaserWriter that, in some cases, have to be addressed in a printer dependent way. This Note describes what the LaserWriter can and cannot do, memory considerations, speed considerations, as well as other things you need to watch out for if you want to make your printing more efficient on the LaserWriter.

How To Determine The Currently Selected Printer

With the addition of new picture comments and the `PrGeneral` procedure, an application should never need to know the type of device to which it is connected. However, some developers feel their application should be able to take advantage of all of the features provided by a particular device, not just those provided by the Printing Manager, and in doing so, these developers produce device-dependent applications, which can produce unpredictable results third-party and new Apple printing devices. For this reason, Apple strongly recommends that you use only the features provided by the Printing Manager, and do not try to use unsupported device features.

Even though there is no supported method for determining a device's type, there is one method described in the original *Inside Macintosh* that still works for ImageWriter and LaserWriter printer drivers. This method is **not** supported, meaning that at some point in the future it will no longer work. If you use this method in your application, it is up to you to weigh the value of the feature against the compatibility risk. The following method works for all ImageWriter, ImageWriter II, and LaserWriter (original, Plus, IINT, IINTX) drivers. Since all new devices released from Apple and third-party developers have their own unique ID, it is up to you to decide what to do with an ID that your application does not recognize.

If you are using the high-level Printing Manager interface, first call `PrValidate` to make sure you have the correct print record. Look at the high byte of the `wdev` word in the `TPrStl` subrecord of the print record. Note that if you have your own driver and want to have your own number, please let DTS know, and DTS can register it.

Following is the current list of printer IDs:

Printer	wDev
ImageWriter I, ImageWriter II	1
LaserWriter, LaserWriter Plus, LaserWriter IINT, LaserWriter IINTX, and Personal LaserWriter NT	3
LaserWriter IISC, Personal LaserWriter SC	4
ImageWriter LQ	5

If you are using the low-level Printing Manager interface, there is no dependable way of getting the wDev information. You should **not** attempt to determine the device ID when using the low-level Printing Manager interface.

Using QuickDraw With the LaserWriter

When you print to the LaserWriter, all of the QuickDraw calls you make are translated (via QuickDraw bottlenecks) into PostScript®, which is in the LaserWriter ROM. Most of the operations available in QuickDraw are available in PostScript, with a few exceptions. The LaserWriter driver does not support the following:

- `XOR` and `NotXOR` transfer modes.
- The grafverb `invert`.
- `_SetOrigin` calls within `PrOpenPage` and `PrClosePage` calls. Use `OffsetRect` instead. (This is fixed in version 3.0 and later of the driver.)
- Regions are ignored. You can simulate regions using polygons or bitmaps. Refer to Technical Note #41, Drawing Into An Off-Screen Bitmap, for how to create off-screen bitmaps.
- Clip regions should be limited to rectangles.
- There is a small difference in character widths between screen fonts and printer fonts. Only the end points of text strings are the same.

What You See Is Not Always What You Get

Unfortunately, what you see on the screen is not always what you get. If you are using standard graphic objects, like rectangles, circles, etc., the object is the same size on the LaserWriter as it is on the screen. There are, however, two types of objects where this is not the case: text and bitmaps.

The earlier noted difference between the widths of characters on the screen and the widths of characters on the printer is due to the difference in resolution. However, to maintain the integrity of line breaks, the driver changes the word and character spacing to maintain the end points of the lines as specified. What this all means is that you cannot count on the positions or the widths of printed characters being exactly the same as they are on the screen. This is why in the original MacDraw®, for example, if one carefully places text and a rectangle and prints it, the text sometimes extends beyond the bounds of the rectangle on the printed page. If an application does its own line layout (i.e., positions the words on the line itself), then it may want to disable the LaserWriter's line layout routines. To disable these routines, use the `LineLayoutOff` picture comment described in the LaserWriter Reference Manual and Technical Note #91, Optimizing for the LaserWriter—Picture Comments.

The sole exception to this rule is if an application is running on 128K ROMs or later. The 128K ROM Font Manager supports the specification of fractional pixel widths for screen fonts, increasing the screen to printer accuracy. This fractional width feature is disabled by default. To enable it, an application can use `_SetFractEnable`, after calling `_InitFonts`.

Applications can use picture comments to left-, right-, or center-justify text. Only the left, right, or center end points are accurate. If the text is fully justified, both end points are accurate. Technical Note #91, Optimizing for the LaserWriter—Picture Comments, discusses these picture comments.

Memory Considerations

To print to the LaserWriter, you need to make sure that you have enough memory available to load the driver's code. The best way to do this is to have all the code you need for printing in a separate segment and unload everything else. When you print to the LaserWriter you are only able to print in Draft mode. You are not able to spool (as the ImageWriter does in the standard or high-quality settings), and your print code, data, and the driver code have to be resident in memory.

In terms of memory requirements, there is not any magic number that always works with all printer drivers (including third-party printer drivers) that are available for the Macintosh. To make sure there is enough memory available during print time, you should make your printing code a separate segment and swap out all unwanted code and data before you call `_PrOpen`.

Printable Paper Area

On the LaserWriter there is a 0.45-inch border that surrounds the printable area of the paper (this is assuming an 8.5" x 11" paper). If you select the "Larger Print Area" option in the Page Setup dialog box, the border changes to 0.25 of an inch. This printable area is different than the available print area of the ImageWriter. An application cannot print a larger area because of the memory PostScript needs to image a page. PostScript takes the amount of memory available in the printer and centers it on the paper, and there is not enough RAM in the LaserWriter to image an entire sheet of paper.

Page Sizes

Many developers have expressed a desire to support page sizes other than those provided by the Apple printer drivers. Even though some devices can physically support other page sizes, there is no way for an application to tell the driver to use this size. With the ImageWriter driver, it is possible to modify certain fields in the print record and expand the printable area of the page. However, each of the Apple drivers implements the page sizes in a different way. No one method works for all drivers. Because of this difference, it is strongly recommended that applications do not attempt to change the page sizes provided in the "Style" dialog box. If your application currently supports page sizes other than those provided by the printer driver, you are taking a serious compatibility risk with future Apple and third-party printer drivers.

Speed Considerations

Although the LaserWriter is relatively fast, there are some techniques an application can use to ensure its maximum performance.

- Try to avoid using the QuickDraw Erase calls (e.g., `_EraseRect`, `_EraseOval`, etc.). It takes a lot of time to handle the erase function because every bit (90,000 bits per square inch) has to be cleared. Erasing is unnecessary because the paper does not need to be erased the way the screen does.

- Printing patterns takes time, since the bitmap for the pattern has to be built. The patterns `black`, `white`, and all the gray patterns have been optimized to use the PostScript gray scales. If you use a different pattern it works, but it just takes longer than usual. In addition, the patterns in driver version 3.0 are rotated; they are not rotated in version 1.0.

- Try to avoid frequently changing fonts. PostScript has to build each character it needs either by using the drawing commands for the built-in LaserWriter fonts or by resizing bitmaps downloaded from screen fonts on the Macintosh. As each character is built, it is cached (if there's room), so if that character is needed again PostScript gets if from the cache. When the font changes, the characters have to be built from scratch in the new font, which takes time. If the font is not in the LaserWriter, it takes time to download it from the Macintosh. If the user has the option of choosing fonts, you have no control over this variable; however, if you control which fonts to use, keep this in mind.

- Avoid using `_TextBox`. It makes calls to `_EraseRect`, which slows the printer, for every line of text it draws. You might want to use a different method of displaying text (e.g., `_DrawString` or `_DrawText`) or write your own version of `_TextBox`. If an application is currently calling `_TextBox`, changing to another method of displaying text can improve speed on the order of five to one.

- Because of the way rectangle intersections are determined, if your clip region falls outside of the `rPage` rectangle, you slow down the printer substantially. By making sure your clip region is entirely within the `rPage` rectangle, you can get a speed improvement of approximately four to one.

- Do not use spool-a-page/print-a-page as some applications do when printing on the ImageWriter. It slows things down considerably because of all of the preparation that has to be done when a job is initiated. Refer to Technical Note #125, Effect of Spool-A-Page/Print-A-Page on Shared Printers, for more information.

- Using `_DrawChar` to place every character to print can take a lot of time. One reason, of course, is because it has to go through the bottlenecks for every character that is drawn. The other is that the printer driver does its best to do line layout, making the character spacing just right. If you are trying to position characters and the driver is trying to position characters too, there is conflict, and printing takes much longer than necessary. In version 3.0 of the driver, there are picture comments that turn off the line layout optimization, alleviating some of the problem. Refer to Technical Note #91, Optimizing for the LaserWriter—Picture Comments, for more information.

Clipping Within Text Strings

When clipping characters out of a string, make sure that the clipping rectangle or region is greater than the bounding box of the text you want to clip. The reason is that if you clip part of a character (e.g., a descender), the clipped character has to be rebuilt, which takes time. In addition, because of the difference between screen fonts and printer fonts, chances are that you cannot accurately clip the right characters unless you are running on the 128K ROMs and have fractional pixel widths enabled.

When to Validate the Print Record

To validate the print record, call `PrValidate`. You need validation to check to see if all of the fields are accurate according to the current printer selected and the current version of the driver. You should call `PrValidate` when you have allocated a new print record or whenever you need to access information from the print record (i.e., when you get `rPage`). The routines `PrStlDialog` and `PrJobDialog` call `PrValidate` when they are called, so you do not have to worry about it if you use these calls.

Empty QuickDraw Objects

QuickDraw objects that are empty (i.e., they have no pixels in them) and are filled but not framed, do not print on the ImageWriter and do not show up on the screen; however, on the LaserWriter they are real objects and do print.

Further Reference:

- *Inside Macintosh*, Volume I, QuickDraw
- *Inside Macintosh*, Volume II, The Printing Manager
- *LaserWriter Reference Manual*
- Technical Note #41, Drawing Into An Off-Screen Bitmap
- Technical Note #91, Optimizing for the LaserWriter—Picture Comments
- Technical Note #125, Effect of Spool-A-Page/Print-A-Page on Shared Printers
- Technical Note #161, A Printing Loop That Cares...
- PostScript Language Reference, Adobe Systems, Incorporated
- PostScript Language Tutorial and Cookbook, Adobe Systems, Incorporated

MacDraw is a registered trademark of Claris Corporation.
PostScript is a registered trademark of Adobe Systems, Incorporated.

#73: Color Printing

See also: QuickDraw
The Printing Manager
PostScript Language Reference Manual,
Adobe Systems

Written by: Ginger Jernigan February 3, 1986
Modified by: Scott "ZZ" Zimmerman January 1, 1988
Updated: March 1, 1988

This discusses color printing in a Macintosh application.

Whereas the original eight-color model of QuickDraw was sufficient for printing in color on the ImageWriter II, the introduction of Color QuickDraw has created the need for more sophisticated printing methods.

The first section describes using the eight-color QuickDraw model with the ImageWriter II and ImageWriter LQ drivers. Since the current Print Manager does not support Color GrafPorts, the eight-color model is the only method available for the ImageWriters.

The next section describes a technique that can be used for printing halftone images using PostScript (when it is available). Also described is a device independent technique for sending the PostScript data. This technique can be used on any LaserWriter driver 3.0 or later. It will work with all LaserWriters except the the LaserWriter IISC.

It is very likely that better color support will be added to the Print Manager in the future. Until then, these are the best methods available.

Part 1, ImageWriters

The ImageWriter drivers are capable of generating each of the eight standard colors defined in QuickDraw by the following constants:

```
whiteColor
blackColor
redColor
greenColor
blueColor
cyanColor
magentaColor
yellowColor
```

To generate color all you need to do is set the foreground and background colors before you begin drawing (initially they are set to blackColor foreground and whiteColor background). To do this you call the QuickDraw routines `ForeColor` and `BackColor` as described in *Inside Macintosh*. If you are using QuickDraw pictures, make sure you set the foreground and background colors before you call `ClosePicture` so that they are recorded in the picture. Setting the colors before calling `DrawPicture` doesn't work.

The drivers also recognize two of the transfer modes: `srcCopy` and `srcOr`. The effect of the other transfer modes is not well defined and has not been tested. It may be best to stay away from them.

Caveats

When printing a large area of more than one color you will encounter a problem with the ribbon. When you print a large area of one color, the printer's pins pick up the color from the back of the ribbon. When another large area of color is printed, the pins deposit the previous color onto the back of the ribbon. Eventually the first color will come through to the front of the ribbon, contaminating the second color. You can get the same kind of effect if you set, for example, a foreground color of yellow and a background color of blue. The ribbon will pick up the blue as it tries to print yellow on top of it. This problem is partially alleviated in the 2.3 version of the ImageWriter driver by using a different printing technique.

The ribbon goes through the printer rather quickly when printing large areas. When the ribbon comes through the second time the colors don't look too great.

Part 2, LaserWriters

Using the PostScript 'image' Operator to Print Halftones

About 'Image'

The PostScript image operator is used to send Bitmaps or Pixmaps to the LaserWriter. The image operator can handle depths from 1 to 8 bits per pixel. Our current LaserWriters can only image about twenty shades of gray, but the printed page will look like there's more. Being that the image operator is still a PostScript operator, it expects its data in the form of hexidecimal bytes. The bytes are represented by two ASCII characters(0-9,A-F). The image operator takes these parameters:

```
width  height  depth  matrix  image-data
```

The first three are the width, height, and depth of the image, and the matrix is the transformation matrix to be applied to the current matrix. See the *PostScript Language Reference Manual* for more information. The image data is where the actual hex data should go. Instead of inserting the data between the first parameters and the image operator itself, it is better to use a small, PostScript procedure to read the data starting from right after the image operator. For example:

```
640 480 8 [640 0 0 480 0 0]
{currentfile picstr readhexstring pop}
image
FF 00 FF 00 FF 00 FF 00 ...
```

In the above example, the width of the image is 640, the height is 480, and the depth is 8. The matrix (enclosed in brackets) is setup to draw the image starting at QuickDraw's 0,0 (top left of page), and with no scaling. The PostScript code (enclosed in braces) is not executed. Instead, it is passed to the image operator, and the image operator will call it repeatedly until it has enough data to draw the image. In this case, it will be expecting 640*480 bytes. When the image operator calls the procedure, it does the following:

1. Pushes the current file which in this case is the stream of data coming to the LaserWriter over AppleTalk. This is the first parameter to readhexstring.

2. Next picstr is pushed. picstr is a string variable defined to hold one row of hex data. The PostScript to create the picstr is:

   ```
   /picstr 640 def
   ```

3. Now readhexstring is called to fill picstr with data from the current file. It begins reading bytes which are the characters following the image operator.

4. Since readhexstring leaves both the string we want, and a boolean that we don't want on the stack, we do one pop to kill of the boolean. Now the string is left behind for the image operator to use.

So using the above PostScript code you can easily print an image. Just fill in the width height and depth, and send the hex data immediately following the PostScript code.

Setting Up for 'image'

Most of the users of this technique are going to want to print a Color QuickDraw PixMap. Although the image command does a lot of the work for you, there are still a couple of tricks that are recommended for performance.

Assume the Maximum Depth

Since the current version of the image operator has a maximum depth of 8 bits/pixel, it is wise to convert the source image to the same depth before imaging. This can be done very simply by using an offscreen GrafPort that is set to 8 bits/pixel, and then using CopyBits to do the depth conversion for you. This will do a nice job of converting lower resolution images to 8 bits/pixel.

Build a Color Table

An 8 bit deep image can only use 256 colors. Since the image that you are starting with is probably color, and the image you get will be grayscale, you need to convert the colors in the source color table into PostScript grayscale values. This is actually easy to do using the Color Manager. First create a table that can hold 512 bytes. This is 2 bytes for each color value from 0 to 255. Since PostScript wants the values in ASCII, you need two characters for each pixel. Now loop through the colors in the color table. Call Index2Color to get the real RGB color for that index, and then call RGB2HSL to convert the RGB color into a luminance value. This value will be expressed as a SmallFract which can then be scaled into a value from 0 to 255. This value should then be converted to ASCII, and stored at the appropriate location in the table. When you are done, you should be able to use a pixel value as an index into your table of PostScript color values. For each pixel in the image, send two characters to the LaserWriter.

Sending the Data

Once you have set up the color table, all that left to do is to loop through all of the pixels, and send their PostScript representation to the LaserWriter. There are a couple of ways to do this. First is to use the low-level Print Manager interface and stream the PostScript using the stdBuf PrCtlCall. Although this seems like it would be the fastest way, the latest version of the LaserWriter driver (5.0) converts all low-level calls to their high level equivalent before executing them. Because of this, the low-level interface is no longer faster than the high level. In an FKEY I have written, I use the high-level Print Manager interface, and send the data via the PostScriptHandle PicComment. This way, I can buffer a large amount of data, before actually sending it. Using this technique, I have been able to image a Mac II screen in about 5 minutes on a LaserWriter Plus, and about 1.5 minutes on a LaserWriter II NTX.

#74: Don't Use the Resource Fork for Data

See also: The Resource Manager
Technical Note #62—Resource Header Application Bytes

Written by: Bryan Stearns March 13, 1986
Updated: March 1, 1988

Don't use the resource fork of a file for non-resource data. Parts of the system (including the File Manager and the Finder) assume that if this fork exists, it will contain valid Resource Manager information.

`PBOpenRF` was provided to allow copying of the resource fork of a file in its entirety, without Resource Manager interpretation. Do not use it to open "another data fork."

The File Manager assumes that the first block of the resource fork of a file will be part of the resource header, and puts information there to aid in scavenging. Note that this means that if you copy a resource file (opened with `PBOpenRF`), the duplicate may not be exactly like the original.

Macintosh
Technical Notes



Developer Technical Support

#75: Apple's Multidisk Installer

Revised by: Jon Zap
Written by: Scott Douglass

April 1991
March 1986

This Technical Note documents Apple's Multidisk Installer, and it is in addition to separate Installer documentation which provides the details of writing scripts.
Changes since February 1991: Revised information about where to find the Multidisk Installer reference suite.

Apple's Multidisk Installer is intended to make it easy for Macintosh users to add or update software. It is a very useful tool for adding third-party software, and Apple recommends that you use the Installer unless your software installation is simple. Apple also recommends that you use version 3.1 of the Installer until version 3.2 is available.

The Multidisk Installer has the following features, as of version 3.1:

- "Easy Install" mode where the Installer script writer can determine the appropriate installation based upon examination of the target environment and provide the user with "One-Button Installation"
- An optional "Custom Install" mode where power users can customize their installation
- "Live" installation to the currently booted and active System; this means it is no longer necessary to ship the Installer on a bootable disk with a System Folder
- Ability to install from an AppleShare server ("Network Install")
- Ability to install from multiple source disks
- Installation of software to folders other than the System Folder as well as creation of new folders as necessary
- Runs under System 4.2 and later versions
- "User Function" support; this feature provides linkage to developer-defined code segments during Easy Install, so script writers can customize the process of determining what software to install and how to install it
- "Action Atom" support; this feature provides linkage to developer-defined code segments which are called before or after the installation takes place; script writers can use this feature to extend the capabilities of the Installer
- Audit Records; this feature provides the script writer with the ability to record details about an installation so that future installations can be more intelligent

The `'indm'` (default map) resource of Installer 3.0.1 is no longer supported in Installer 3.1 and later versions. This was used by script writers to implement Easy Install. It is replaced by `'infr'` (framework) and `'inrl'` (rule) resources.

Note: If the user opens the Installer document rather than the Installer, the wrong Installer may be launched (depending upon the contents of their mounted volumes). This is only a problem between versions 3.1 and 3.0.x. If you are developing a 3.1 script, you may want to add an `'indm'` resource that puts up a warning dialog box. If

---

you are developing a 3.0.x script, you may want to add an `'infr'` and `'inrl'` resource that puts up a `reportSysError` dialog box. This problem is resolved in Installer 3.2.

Multidisk Installer version 3.1 is available as a complete reference suite which includes the following:

- *Installer 3.1 Scripting Guide* (dated December 17,1990 on the cover)
- *ScriptCheck User's Manual*
- Installer 3.1 application
- ScriptCheck 2.0b6 (MPW Tool)
- InstallerTypes.r (MPW Rez interface file)

The reference suite for Multidisk Installer 3.1 is available on the Developer CD series (note that documentation on Developer CD V (*Night of the Living Disc*) and Developer CD VI (*Gorillas in the Disc*) is preliminary and partly inaccurate); AppleLink in the Developer Services Bulletin Board, ETO #3 (available from APDA), the latest System Software 7.0 Beta CD-ROM (not including the original b1 release) distributed to Apple Partners and Associates, and on the Apple FTP site (IP 130.43.2.3) on the Internet.

Multidisk Installer version 3.2 will be released soon. It contains a few minor improvements that will make it easier to write scripts that work on both system 6.0.x and 7.0. Installer 3.1 has had minimal testing with System 7.0. If you are expecting to install software onto machines running System 7.0, then you should consider upgrading. Script changes should be minimal.

Installation of software is a nontrivial process. Apple recommends that you reserve time for development and testing to ensure that the installation process proceeds smoothly on all target machine configurations.

To ship the Installer with your product you must contact Apple's Software Licensing Department (AppleLink: SW.LICENSE) and license the Installer alone or with the System Software package that includes the version of the Installer you intend to use. Software Licensing also supplies you with a copy of the Installer that you may ship.

Further Reference:

- *Multidisk Installer 3.1 Suite*

#76: The Macintosh Plus Update Installation Script

Written by: scott douglass
Updated:

February 24, 1986
March 1, 1988

Earlier versions of this note described the Macintosh Plus Update installation script, because it was the first script created for the Installer. Since then, many versions of this script have been created which no longer match what was described here. In addition, many other scripts now exist.

#77: HFS Ruminations

See also: The File Manager
Technical Note #66—
Determining Which File System is Active
Technical Note #67—Finding the "Blessed Folder"
Technical Note #68—
Searching All Directories on an HFS Volume

Written by: Jim Friedlander June 7, 1986
Updated: March 1, 1988

This technical note contains some thoughts concerning HFS.

HFS numbers

A drive number is a small positive word (e.g. 3).

A `VRefNum` (as opposed to a `WDRefNum`) is a small negative word (e.g. `$FFFE`).

A `WDRefNum` is a large negative word (e.g. `$8033`).

A `DirID` is a long word (e.g. 38). The root directory of an HFS volume always has a `dirID` of 2.

Working Directories

Normally an application doesn't need to open working directories (henceforth `WDs`) using `PBOpenWD`, since `SFGetFile` returns a `WDRefnum` if the selected file is in a directory on a hierarchical volume and you are running HFS. There are times, however, when opening a `WD` is desirable (see the discussion about `BootDrive` below).

If you do open a `WD`, it should be created with an `ioWDProcID` of 'ERIK' (`$4552494B`) and it will be deallocated by the Finder. Note that under MultiFinder, ioWDProcID will be ignored, so you should only use 'ERIK'.

`SFGetFile` also creates `WDs` with an `ioWDProcID` of 'ERIK'. If `SFGetFile` opens two files from the same directory (during the same application), it will only create one working directory.

There are no WDRefnums that refer to the root—the root directory of a volume is always referred to by a vRefNum.

When you can use HFS calls

All of the HFS 'H' calls, except for PBHSetVInfo, can be made without regard to file system as long as you pass in a pointer to an HFS parameter block. PBHGetVol, PBHSetVol (see the warnings in the File Manager chapter of *Inside Macintosh*), PBHOpen, PBHOpenRF, PBHCreate, PBHDelete, PBHGetFInfo, PBHSetFInfo, PBHSetFLock, PBHRstFLock and PBHRename differ from their MFS counterparts only in that a dirID can be passed in at offset $30.

The only difference between, for example, PBOpen and PBHOpen is that bit 9 of the trap word is set, which tells HFS to use a larger parameter block. MFS ignores this bit, so it will use the smaller parameter block (not including the dirID). Remember that all of these calls will accept a WDRefNum in the ioVRefNum field of the parameter block.

PBHGetVInfo returns more information than PBGetVInfo, so, if you're counting on getting information that **is** returned in the HFS parameter block, but not in the MFS parameter block, you should check to see which file system is active.

HFS-specific calls can only be made if HFS is active. These calls are: PBGetCatInfo, PBSetCatInfo, PBOpenWD, PBCloseWD, PBGetFCBInfo,PBGetWDInfo, PBCatMove and PBDirCreate. PBHSetVInfo has no MFS equivalent. If any of these calls are made when MFS is running, a system error will be generated. If PBCatMove or PBDirCreate are called for an MFS volume, the function will return the error code –123 (wrong volume type). If PBGetCatInfo or PBSetCatInfo are called on MFS volumes, it's just as if PBGetFInfo and PBSetFInfo were called.

Default volume

If HFS is running, a call to GetVol (before you've made any SetVol calls) will return the WDRefNum of your application's parent directory in the vRefNum parameter. If your application was launched by the user clicking on one or more documents, the WDRefNums of those documents' parent directories are available in the vRefNum field of the AppFile record returned from GetAppFiles.

If MFS is running, a call to GetVol (before you've made any SetVol calls) will return the vRefNum of the volume your application is on in the vRefNum parameter. If your application was launched by the user clicking on one or more documents, the vRefNum of those documents' volume are available in the vRefNum field of the AppFile record returned from GetAppFiles.

BootDrive

If your application or desk accessory needs to get the `WDRefNum` of the "blessed folder" of the boot drive (for example, you might want to store a configuration file there), it can not rely on the low-memory global `BootDrive` (a word at `$210`) to contain the correct value. If your application is the startup application, `BootDrive` will contain the `WDRefNum` of the directory/volume that your application is in (not the `WDRefNum` of the "blessed folder"); Your application could have been `_Launched` from an application that has modified `BootDrive`; if you are a desk accessory, you might find that some applications alter `BootDrive`.

To get the "real" `WDRefNum` of the "blessed folder" that contains the currently open System file, you should call SysEnvirons (discussed in Technical Note #129). If that is impossible, you can do something like this (**Note:** if you are running under MFS, `BootDrive` always contains the `vRefNum` of the volume on which the currently open System file is located):

```
...
CONST
        SysWDProcID    = $4552494B; {"ERIK"}
        BootDrive      = $210;      {address of Low-Mem global BootDrive}
        FSFCBLen       = $3F6;      {address of Low-Mem global to
                                     distinguish file systems }
        SysMap         = $A58;      {address of Low-Mem global that contains
                                     system map reference number}

TYPE
        WordPtr = ^Integer;                 {Pointer to a word(2 bytes)}
...

FUNCTION HFSExists: BOOLEAN;

Begin {HFSExists}
        HFSExists := WordPtr(FSFCBLen)^ > 0;
End;  {HFSExists}


FUNCTION GetRealBootDrive: INTEGER;

VAR
        MyHPB          : HParamBlockRec;
        MyWDPB         : WDPBRec;
        err            : OSErr;
        sysVRef        : integer; {will be the vRefNum of open system's vol}

Begin {GetRealBootDrive}
        if HFSExists then Begin    {If we're running under HFS... }

                {get the VRefNum of the volume that }
                {contains the open System File      }
                err:= GetVRefNum(WordPtr(SysMap)^,sysVRef);
```

```
                with MyHPB do Begin
                {Get the "System" vRefNum and "Blessed" dirID}
                        ioNamePtr    := NIL;
                        ioVRefNum    := sysVRef; {from the GetVrefNum call}
                        ioVolIndex   := 0;
                End; {with}
                err := PBHGetVInfo(@MyHPB, FALSE);

                with myWDPB do Begin        {Open a working directory there}
                        ioNamePtr    := NIL;
                        ioVRefNum    := sysVRef;
                        ioWDProcID   := SysWDProcID; {Using the system proc ID}
                        ioWDDirID    := myHPB.ioVFndrInfo[1];{ see TechNote 67}
                End; {with}
                err := PBOpenWD(@myWDPB, FALSE);

                GetRealBootDrive := myWDPB.ioVRefNum;
                {We've got the real WD}
        End Else {we're running MFS}
                GetRealBootDrive := WordPtr(BootDrive)^;
                {BootDrive is valid under MFS}
End;  {GetRealBootDrive}
```

From MPW C:

```
/*"ERIK"*/
#define     SysWDProcID 0x4552494B
#define     BootDrive   0x210
/*address of Low-Mem global that contains system map reference number*/
#define     SysMap      0xA58
#define     FSFCBLen    0x3F6
#define     HFSIsRunning ((*(short int *)(FSFCBLen)) > 0)

OSErr GetRealBootDrive(BDrive)
short int *BDrive;
{ /*GetRealBootDrive*/

      /*three different parameter blocks are used here for clarity*/
      HVolumeParam      myHPB;
      FCBPBRec          myFCBRec;
      WDPBRec           myWDPB;
      OSErr             err;
      short int         sysVRef; /*will be the vRefNum of open system's
                                   vol*/

      if (HFSIsRunning)

      { /*if we're running under HFS... */

      /*get the vRefNum of the volume that contains the open System File*/
            myFCBRec.ioNamePtr= nil;
            myFCBRec.ioVRefNum = 0;
            myFCBRec.ioRefNum = *(short int *)(SysMap);
            myFCBRec.ioFCBIndx = 0;

            err = PBGetFCBInfo(&myFCBRec,false);
            if (err != noErr) return(err);
      /*now we need the dirID of the "Blessed Folder" on this volume*/
```

```
            myHPB.ioNamePtr = nil;
            myHPB.ioVRefNum = myFCBRec.ioFCBVRefNum;
            myHPB.ioVolIndex = 0;

            err = PBHGetVInfo(&myHPB,false);
            if (err != noErr) return(err);

        /*we can now open a WD for the directory that contains the open
        system file one will most likely already be open, so PBOpenWD will
        just return that WDRefNum*/
            myWDPB.ioNamePtr = nil;
            myWDPB.ioVRefNum = myHPB.ioVRefNum;
            myWDPB.ioWDProcID = SysWDProcID; /*'ERIK'*/
            myWDPB.ioWDDirID = myHPB.ioVFndrInfo[0]; /* see Technote # 67
                                                  [c has 0-based arrays]*/

            err = PBOpenWD(&myWDPB,false);
            if (err != noErr) return err;

            *BDrive = myWDPB.ioVRefNum; /*that's all!*/
        } /* if (HFSIsRunning) */
        else
            *BDrive = *(short int *)(BootDrive);
            /*BootDrive is valid under MFS*/

        return noErr;
    }                                       /*GetRealBootDrive*/
```

The Poor Man's Search Path (PMSP)

If HFS is running, the PMSP is used for any file system call that can return a file-not-found error, such as `PBOpen`, `PBClose`, `PBDelete`, `PBGetCatInfo`, etc. It is **not** used for indexed calls (that is, where `ioFDirIndex` is positive) or when a file is created (`PBCreate`) or when a file is being moved between directories (`PBCatMove`). The PMSP is also **not** used when a non-zero `dirID` is specified.

Here's a brief description of how the default PMSP works.

1) The directory that you specify (specified by `WDRefNum` or pathname) is searched; if the specified file is not found, then

2) the volume/directory specified by `BootDrive` (low-memory global at `$210`) is searched IF it is on the same volume as the directory you specified (see #1 above); if the specified file is not found, or the directory specified by `BootDrive` is not on the same volume as the directory that you specified, then

3) if there is a "blessed folder" on the same volume as the directory you specified (see #1 above), it is searched. Please note that if #2 above specifies the same directory as #3, then that directory is **not** searched twice. If no file is found, then

4) `fnfErr` is returned.

ioDirId and ioFlNum

Two fields of the HParamBlockRec record share the same location. ioDirID and ioFlNum are both at offset $30 from the start of the parameter block. This causes a problem, since, in some calls (e.g. PBGetCatInfo), a dirID is passed in and a file number is returned in the same field.

Future versions of Apple's HFS interfaces will omit the ioFlNum designator, so, if you need to get the file number of a file, it will be in the ioDirID of the parameter block **after** you have made the call. If you are making successive calls that depend on ioDirID being set correctly, you must "reset" the ioDirID field before each call. The program fragment in Technical Note #68 does this.

PBHGetVInfo

Normally, PBHGetVInfo will be called specifying a vRefNum. There are times, however, when you may make the call and only specify a volume name. If this is so, there are a couple of things to look out for.

Let's say that we have two volumes mounted: "Vol1:" (the default volume) and "Vol2:". We also have a variable of type HParamBlockRec called MyHPB. We want to get information about Vol2:, so we put a pointer to a string (let's call it fName) in MyHPB.ioNamePtr. The string fName is equal to "Vol2" (Please note the missing colon). We also initialize MyHPB.ioVRefNum to 0. Then we make the call. We are very surprised to find out that we are returned an error of 0 (noErr) and that the ioVRefNum that we get back is **not** the vRefNum of Vol2:, but rather that of Vol1:.

Here's what's happening: PBHGetVInfo looks at the volume name, and sees that it is improper (it is missing a colon). So, being a forgiving sort of call, it goes on to look at the ioVRefNum field that you passed it (see pp. 99 of *Inside Macintosh*, vol. II). It sees a 0 there, so it returns information about the default volume.

If you want to get information about a volume, and you just have its name and you are not sure that the name is a proper one, you should set MyHPB.ioVRefNum to –32768 ($8000). No vRefNum or WDRefNum can be equal to $8000. By doing this, you are forcing PBHGetVInfo to use the volume name and, if that name is invalid, to return a –35 error (nsvErr), "No such volume."

PBGetWDInfo and Register D1

There was a problem with PBGetWDInfo that sometimes caused the call to inaccurately report the dirID of a directory. It is fixed in System 3.2 and later. To be absolutely sure that you won't get stung by this, clear register D1 (CLR.L D1) before a call to PBGetWDInfo. You can do this either with an INLINE (Lisa Pascal and most C's) or with a short assembly-language routine before the call to PBGetWDInfo.

#78: Resource Manager Tips

See also: The Resource Manager
The Memory Manager
The Menu Manager
Technical Note #129—SysEnvirons

Written by: Jim Friedlander June 8, 1986
Updated: March 1, 1988

This note discusses some problems with the Resource Manager and how to work around them.

OpenResFile Bug

This section of the note formerly described a bug in `OpenResFile` on 64K ROM machines. Information specific to 64K ROM machines has been deleted from Macintosh Technical Notes for reasons of clarity.

GetMenu and ResErrProc

If your application makes use of `ResErrProc` (a pointer to a procedure stored in low-memory global `$AF2`) to detect resource errors, you will get unexpected calls to your `ResErrProc` procedure when calling `GetMenu` on 128K ROMs. The Menu Manager call `GetMenu` makes a call to `GetResInfo`, requesting resource information about MDEF 0. Unfortunately, `ROMMapInsert` is set to `FALSE`, so this call fails, setting `ResErr` to –192 (`resNotFound`). This in turn will cause a call to your `ResErrProc`, procedure even though the `GetMenu` call has worked correctly. This is **only** a problem if you are using `ResErrProc`.

The workaround is to:

1) save the address of your `ResErrProc` procedure
2) clear `ResErrProc`
3) do a `GetResource` call on the MENU resource you want to get
4) check to see if you get a nil handle back, if you do, you can handle the error in whatever way is appropriate for your application
5) call `GetMenu`, and
6) when you are done calling `GetMenu`, restore `ResErrProc`

SetResAttrs on read-only resource maps

SetResAttrs does not return an error if you are setting the resource attributes of a resource in a resource file that has a read-only resource map. The workaround is to check to see if the map is read-only and proceed from there:

```
CONST
    MapROBit = 8; {Toolbox bit ordering for bit 7 of low-order byte}

BEGIN
    ...
    attrs:= GetResFileAttrs(refNum);
    IF BitTst(@attrs,MapROBit) THEN ... {write-protected map}
```



#79: _ZoomWindow

Revised by: Craig Prouse — April 1990
Written by: Jim Friedlander — June 1986

This Technical Note contains some hints about using `_ZoomWindow`.
Changes since February 1990: Fixed a bug in `DoWZoom` which caused crashes if the content of a window did not intersect with any device's `gdRect`. Also made `DoWZoom` more robust by making `savePort` a local variable and checking for off-screen and inactive `GDevice` records. (One variable name has changed.) Additional minor changes: Corrected original sample code to use `_EraseRect` before zooming and added references to Human Interface Note #7, Who's Zooming Whom? for more subtle and application-specific considerations.

Basics

`_ZoomWindow` allows a window to be toggled between two states (where "state" means size and location): a default state and a user-selectable state. The default state stays the same unless the application changes it, while the user-selectable state is altered when the user changes the size or location of a zoomable window. The code to handle zoomable windows in a main event loop would look something like the examples which follow.

Note: `_ZoomWindow` assumes that the window that you are zooming is the current `GrafPort`. If `thePort` is not set to the window that is being zoomed, an address error is generated.

MPW Pascal

```
CASE myEvent.what OF
  mouseDown: BEGIN
    partCode:= FindWindow(myEvent.where, whichWindow);
      CASE partCode OF
        inZoomIn, InZoomOut:
          IF TrackBox(whichWindow, myEvent.where, partCode) THEN
            BEGIN
              GetPort(oldPort);
              SetPort(whichWindow);
              EraseRect(whichWindow^.portRect);
              ZoomWindow(whichWindow, partCode, TRUE);
              SetPort(oldPort);
            END; {IF}
        ...{and so on}
      END; {CASE}
    END; {mouseDown}
  ...{and so on}
END; {CASE}
```

MPW C

```
switch (myEvent.what) {
  case mouseDown:
    partCode = FindWindow(myEvent.where, &whichWindow);
    switch (partCode) {
      case inZoomIn:
      case inZoomOut:
        if (TrackBox(whichWindow, myEvent.where, partCode)) {
          GetPort(&oldPort);
          SetPort(whichWindow);
          EraseRect(whichWindow->portRect);
          ZoomWindow(whichWindow, partCode, true);
          SetPort(oldPort);
          } /* if */
        break;
      ... /* and so on */
    } /* switch */
  ... /* and so on */
} /* switch */
```

If a window is zoomable, that is, if it has a window definition `ID = 8` (using the standard `'WDEF'`), `WindowRecord.dataHandle` points to a structure that consists of two rectangles. The user-selectable state is stored in the first rectangle, and the default state is stored in the second rectangle. An application can modify either of these states, though modifying the user-selectable state might present a surprise to the user when the window is zoomed from the default state. An application should also be careful to not change either rectangle so that the title bar of the window is hidden by the menu bar.

Before modifying these rectangles, an application must make sure that `DataHandle` is not `NIL`. If it is `NIL` for a window with window definition `ID = 8`, that means that the program is not executing on a system or machine that supports zooming windows.

One need not be concerned about the use of a window with window definition `ID = 8` making an application machine-specific—if the system or machine that the application is running on doesn't support zooming windows, `_FindWindow` never returns `inZoomIn` or `inZoomOut`, so neither `_TrackBox` nor `_ZoomWindow` are called.

If `DataHandle` is not `NIL`, an application can set the coordinates of either rectangle. For example, the Finder sets the second rectangle (default state) so that a zoomed-out window does not cover the disk and trash icons.

For the More Adventurous (or Seeing Double)

Developers should long have been aware that they should make no assumptions about the screen size and use `screenBits.bounds` to avoid limiting utilization of large video displays. Modular Macintoshes and Color QuickDraw support multiple display devices, which invalidates the use of `screenBits.bounds` unless the boundary of only the primary display (the one with the menu bar) is desired. When dragging and growing windows in a **multi-screen environment**, developers are now urged to use the bounding rectangle of the `GrayRgn`. In most cases, this is not a major modification and does not add a significant amount of code. Simply define a variable

```
desktopExtent := GetGrayRgn^^.rgnBBox;
```

and use this in place of `screenBits.bounds`. When zooming a document window, however, additional work is required to implement a window-zooming strategy which fully conforms with Apple's Human Interface Guidelines.

One difficulty is that when a new window is created with `_NewWindow` or `_GetNewWindow`, its default `stdState` rectangle (the rectangle determining the size and position of the zoomed window) is set by the Window Manager to be the gray region of the main display device inset by three pixels on each side. If a window has been moved to reflect a position on a secondary display, that window still zooms onto the main device, requiring the user to pan across the desktop to follow the window. The preferred behavior is to zoom the window onto the device containing the **largest portion** of the unzoomed window. This is a perfect example of a case where it is necessary for the application to modify the default state rectangle before zooming.

`DoWZoom` is a Pascal procedure which implements this functionality. It is a good example of how to manipulate both a `WStateData` record and the Color QuickDraw device list. On machines without Color QuickDraw (e.g., Macintosh Plus, Macintosh SE, Macintosh Portable) the `stdState` rectangle is left unmodified and the procedure reduces to five instructions, just like it is illustrated under "Basics." If Color QuickDraw is present, a sequence of calculations determines which display device contains most of the window prior to zooming. That device is considered dominant and is the device onto which the window is zoomed. A new `stdState` rectangle is computed based on the `gdRect` of the dominant `GDevice`. Allowances are made for the window's title bar, the menu bar if necessary, and for the standard three-pixel margin. (Please note that `DoWZoom` only mimics the behavior of the default `_ZoomWindow` trap as if it were implemented to support multiple displays. It does not account for the "natural size" of a window for a particular purpose. See Human Interface Note #7, Who's Zooming Whom?, for details on what constitutes the natural size of a window.) It is not necessary to set `stdState` prior to calling `_ZoomWindow` when zooming back to `userState`, so the extra code is not executed in this case.

`DoWZoom` is too complex to execute within the main event loop as shown in "Basics," but if an application is already using a similar scheme, it can simply add the `DoWZoom` procedure and replace the conditional block of code following

```
IF TrackBox...
```

with

```
DoWZoom(whichWindow, partCode);.
```

Happy Zooming.

```
PROCEDURE DoWZoom (theWindow: WindowPtr; zoomDir: INTEGER);
VAR
  windRect, theSect, zoomRect : Rect;
  nthDevice, dominantGDevice : GDHandle;
  sectArea, greatestArea : LONGINT;
  bias : INTEGER;
  sectFlag : BOOLEAN;
  savePort : GrafPtr;
BEGIN
  { theEvent is a global EventRecord from the main event loop }
  IF TrackBox(theWindow,theEvent.where,zoomDir) THEN
    BEGIN
      GetPort(savePort);
      SetPort(theWindow);
      EraseRect(theWindow^.portRect);    {recommended for cosmetic reasons}

      { If there is the possibility of multiple gDevices, then we  }
      { must check them to make sure we are zooming onto the right }
      { display device when zooming out. }
      { sysConfig is a global SysEnvRec set up during initialization  }
      IF (zoomDir = inZoomOut) AND sysConfig.hasColorQD THEN
        BEGIN
          { window's portRect must be converted to global coordinates }
          windRect := theWindow^.portRect;
          LocalToGlobal(windRect.topLeft);
          LocalToGlobal(windRect.botRight);
          { must calculate height of window's title bar }
          bias :=   windRect.top - 1
              -  WindowPeek(theWindow)^.strucRgn^^.rgnBBox.top;
          windRect.top := windRect.top - bias; {Thanks, Wayne!}
          nthDevice := GetDeviceList;
          greatestArea := 0;
          { This loop checks the window against all the gdRects in the   }
          { gDevice list and remembers which gdRect contains the largest }
          { portion of the window being zoomed. }
          WHILE nthDevice <> NIL DO
            IF TestDeviceAttribute(nthDevice,screenDevice) THEN
              IF TestDeviceAttribute(nthDevice,screenActive) THEN
                BEGIN
                  sectFlag := SectRect(windRect,nthDevice^^.gdRect,theSect);
                  WITH theSect DO
                    sectArea := LONGINT(right - left) * (bottom - top);
                  IF sectArea > greatestArea THEN
                    BEGIN
                      greatestArea := sectArea;
                      dominantGDevice := nthDevice;
                    END;
                  nthDevice := GetNextDevice(nthDevice);
                END; {of WHILE}
          { We must create a zoom rectangle manually in this case. }
          { account for menu bar height as well, if on main device }
          IF dominantGDevice = GetMainDevice THEN
            bias := bias + GetMBarHeight;
          WITH dominantGDevice^^.gdRect DO
            SetRect(zoomRect,left+3,top+bias+3,right-3,bottom-3);
          { Set up the WStateData record for this window. }
          WStateDataHandle(WindowPeek(theWindow)^.dataHandle)^^.stdState := zoomRect;
        END; {of Color QuickDraw conditional stuff}

      ZoomWindow(theWindow,zoomDir,TRUE);
      SetPort(savePort);
    END;
END;
```

In an attempt to avoid declaring additional variables, the original version of this document was flawed. In addition, the assignment statement responsible for setting the `stdState` rectangle is relatively complex and involves two type-casts. The following may look like C, but it really is Pascal. Trust me.

```
WStateDataHandle(WindowPeek(theWindow)^.dataHandle)^^.stdState := zoomRect;
```

It could be expanded into a more readable form such as:

```
VAR
   theWRec : WindowPeek;
   zbRec   : WStateDataHandle;

theWRec := WindowPeek(theWindow);
zbRec := WStateDataHandle(theWRec^.dataHandle);
zbRec^^.stdState := zoomRect;
```

Further Reference:

- *Inside Macintosh*, Volume IV, The Window Manager (pp. 49–52)
- *Inside Macintosh*, Volume V, Graphics Devices (p. 124), The Window Manager (p. 210)
- Human Interface Note #7, Who's Zooming Whom?



#80: Standard File Tips

See also: The Standard File Package

Written by: Jim Friedlander
Updated:

June 7, 1986
March 1, 1988

SFSaveDisk and CurDirStore

Low-memory location `$214` (`SFSaveDisk`—a word) contains –1 * the `vRefNum` of the volume that SF is displaying (MFS and HFS). It never contains –1 * a `WDRefNum`.

Low-memory location `$398` (`CurDirStore`—a long word) contains the `dirID` of the directory that SF is displaying (HFS only).

This information can be particularly useful at hook time, when the `vRefNum` field of the reply record has not yet been filled in. **Note:** `reply.fName` is filled in correctly at hook time if a file has been selected. If a directory has been selected, `reply.fType` is non-zero (it contains the `dirID` of the selected directory). If neither a file nor a directory is selected, both `reply.fName[0]` and `reply.fType` are 0.

Setting Standard File's default volume and directory

If you want `SFGetFile` or `SFPutFile` to display a certain volume when it draws its dialog, you can put –1 * the `vRefNum` of the volume you wish it to display into the low-memory global `SFSaveDisk` (a word at `$214`).

In Pascal, you would use something like:

```
...
TYPE
   WordPtr = ^INTEGER;                {pointer to a two-byte location}
CONST
   SFSaveDisk = $214;                 {location of low-memory global}
VAR
   SFSaveVRef : WordPtr;
   myVRef     : INTEGER;
BEGIN
...
{myVRef gets assigned here}
...
SFSaveVRef := WordPtr(SFSaveDisk);    {point to SFSaveDisk}
SFSaveVRef^:= -1 * myVRef;            {"stuff" the value in}
SFGetFile(...
```

In C you would use something like this (where a variable of type "short" occupies 2 bytes):

```
#define SFSaveDisk (*(short *)0x214)

short myVRef;
...
/* myVRef gets assigned here */
...
SFSaveDisk = -1 * myVRef; /* "stuff" the value in */
SFGetFile(...
```

If you are running HFS and would like to have Standard File display a particular directory as well as a particular volume, you can't just put a `WDRefNum` into `SFSaveDisk`. If you do put a `WDRefNum` into `SFSaveDisk`, Standard File will display the root directory of the default volume. Instead, you must put –1 * the `vRefNum` into `SFSaveDisk` (see above) and put the `dirID` of the directory that you wish to have displayed in `CurDirStore`. If you put an invalid `dirID` into `CurDirStore`, Standard File will display the root level of the volume referred to by `SFSaveDisk`. To change `CurDirStore` you can use a technique similar to the above, but remember that `CurDirStore` is a four-byte value. If your application is running under MFS, Standard File ignores `CurDirStore`, so you can use the same code regardless of file system.

#81: Caching

See also: The File Manager
The Device Manager
Technical Note #14—The INIT 31 Mechanism

Written by: Rick Blair June 17, 1986
Updated: March 1, 1988

This technical note describes disk and File System caching on the Macintosh, with particular emphasis on the high-level File System cache. Of the three caches used for file I/O, this is the one which could have the most impact on your program. **Note**: This big File System cache is not available on 64K ROM machines.

A term

In this note I will use the term "HFS" to mean the Hierarchical File System **and** the Sony driver which can access the 800K drives. Both RAM-based HFS (Hard Disk 20 file) and the 128K ROM version include the second-generation Sony driver.

There's always a cache (type 1)

The first type of cache used by the File System has been around since the days of the Macintosh File System. Under MFS, each volume has a one-block buffer for all file/volume data. This prevents a read of two bytes followed by a read (at the next file position) of 4 bytes from causing actual disk I/O. The volume allocation map also gets saved in the system heap but it's not really part of the cache.

This type of caching is still used by HFS, which includes MFS-format volumes which may be mounted while running HFS. With HFS, the cache is a little bigger: each volume gets 1 block of buffering for the bitmap, 2 blocks for volume (including file) data, and 16 blocks for HFS B*-tree control buffering.

This cache lives in the system heap (unless HFS is using the new File System caching mechanism, in which case things become more complicated. See "type 3" below).

Cache track fever (type 2)

The track cache, only present with the enhanced Sony driver, will cache the current track (up to twelve blocks) so that subsequent reads to that track may use the cache. The track cache is "write through"; all writes go to both the cache and the Sony disk so flushing is never required.

Track caching only takes place for synchronous I/O calls; when an application makes asynchronous calls it expects to use the time while the disk is seeking, etc. to execute other code.

The track cache gets its storage space from the system heap.

Cache me if you can (type 3)

The last type of cache to be discussed is only available under the 128K and greater ROMs. This user-controlled cache is **not** "write-through".

Based on how much space the user has allocated via the control panel, the File System will set up a cache which can accommodate a certain number of blocks. This storage will come from the application heap in the space above `BufPtr` (see technical note #14 and below). This is really the space above the jump table and the "`A5` world", not technically part of the application heap. However, moving `BufPtr` down will cause a corresponding reduction in the space available to the application heap.

The installation code will also grab the space used by the old File System cache (type 1) since all types of disk blocks can be accommodated by this new cache.

The bulk of the caching **code** used for this RAM cache is also loaded above `BufPtr` at application launch time. This is accomplished by the INIT 35 resource which is installed in the system heap and initialized at boot time. At application launch time, INIT 35 checks the amount of cache allocated via the control panel and moves `BufPtr` down accordingly before bringing in the balance of the caching code. The RAM caching code is in the 'CACH' 1 resource in the System File.

The caching code always makes sure there is room for 128K of application heap and 32K of cache. If the user-requested amount would reduce the heap/cache below these values then the cache space is readjusted accordingly.

Up to 36 separate files may be buffered by the cache. Each queue is a list of blocks cached for that file. Information is kept about the "age" of each block and the blocks are also kept in a list in the order in which they occur in the file. The aging information tells which blocks were least recently used; these are the first to be released when new blocks become eligible for caching. The file order information is useful for flushing the cache to the disk in an efficient manner, i.e. the file order approximates disk order.

Assuming this cache has been enabled by the user, all files which are read from *or* written to by File System (HFS) calls are subject to caching under the current implementation. The cache is not "write through" like the track cache. When a File System write (`PBWrite`, `WriteResource`, etc.) is done, the block is buffered until the block is released (age discrimination), a volume flush is done or the application terminates.

It may be useful to an application to prevent this process of reading and writing "in place". The Finder disables caching of newly read/written blocks while doing file copies since it would be silly to cache files that the Finder was reading into memory anyway. Copy protection schemes may also need this capability. Disabling reading and writing in place is accomplished by setting a bit in a low memory flag byte, `CacheCom` (see below). When you set this flag, no new candidates for caching will be accepted. Blocks already saved may still be read from the cache, of course.

`CacheCom` is at `$39C`. Bit 7 is the bit to set to disable subsequent caching, as follows:

```
        MOVE.B CacheCom,saveTemp ;save away the old value
        BSET.B #7,CacheCom       ;tell caching code to stop R/W I.P.
        ...
        BTST.B #7,saveTemp       ;check saved value
        BNE.S  @69
        BCLR.B #7,CacheCom       ;clear it if it was cleared before
    @69
```

Bit 6 contains another flag which can force **all** I/O to go to the disk. If that flag is set then every time even one byte is requested from the File System the disk will be hit. I can think of no good reason to use this except to test the system code itself. The other bits should likewise be left alone.

Please don't use this feature unnecessarily; the user should retain control over caching. **Important:** if your program doesn't have enough space to run due to caching you should ask the user to disable (or reduce) it with the control panel and then relaunch your application. This may be the subject of a future technical note.

BufPtr

The RAM-resident caching software arbitrates `BufPtr` in the friendliest manner possible. It saves the old value away before changing it, and then when it is time to release its space it looks at it again. If `BufPtr` has been moved again, it knows that it can't restore the old value it saved until `BufPtr` is put back to where it left it. In this manner any subsequent code or data put up under `BufPtr` is assured of not being obliterated by the caching routines.

A final note

To avoid problems with data in the cache not getting written out to disk, call `FlushVol` after each time you write a file to disk. This ensures that the cache is written, in case a crash occurs soon thereafter.

Macintosh Technical Notes

#82: TextEdit: Advice & Descent

See also: TextEdit
Technical Note #22—TEScroll Bug
Technical Note #127—TextEdit EOL Ambiguity
Technical Note #131—TextEdit Bugs

Written by: Rick Blair June 21, 1986
Updated: March 1, 1988

This technical note will point out some bugs (and possible workarounds), and other items of interest for the TextEdit programmer.

TESelRect

Multiple line selections are often more complex shapes than simple rectangles. If this is the case, the `teSelRect` field of the `TERec` is set to the **last** (bottommost) rectangle in the selection. The `teHiHook` is called to invert each line of the selection.

The ROM limits the selection range (i.e. the lines that get set into `teSelRect`) to only those lines which will fit into the `viewRect`. This means that `teSelRect` will be left at the last **visible** line. (The old 64K ROMs made all the calls for the complete selection and just let clipping take care of the rest.)

TEDoText

The parameters of this special hook into TextEdit need a little additional explanation. `D3` and `D4` are described on page 391 of *Inside Macintosh Volume I* as being the first and last characters to be redrawn. This is true but specific to the –1 "`DoDraw`" case. In fact, all the calls to `TEDoText` are interested in these first and last character positions. They determine the selection for a (1) highlight call, the caret position for a (–2) `DoCaret` call (where `D4` is ignored as it's assumed to equal `D3`), etc.

Note that the `DoCaret` (–2) call behaves differently than described in *Inside Macintosh*, as well. Good old page 391 says it sets up the pen position for caret drawing. Since an `InvertRect` call is used to draw the caret if you use the default `teCarHook`, the ROMs just set up `teSelRect`, they don't bother with the QuickDraw pen.

TEScrpLength

Inside Macintosh describes `TEScrpLength` as a long integer; indeed, four bytes are reserved for this value with the intent of someday using that range of values. However, the ROMs use word operations in their accesses to `TEScrpLength` and make word calculations with it. This means that the high word of `TEScrpLength` is used for calculations. This is something to watch out for.

CharWidth

Inside Macintosh says that `CharWidth` takes stylistic variations into account when determining the width of a character. In fact, for *italic* and outlined styles the extra width is not taken into account. TextEdit relies on `CharWidth` for positioning of the caret, etc. If you have chosen to use, for instance, italic style in your TE record you will find that as you type the caret actually overlaps the character to the left and so when the caret is erased some of that character will get erased, too. This is somewhat disconcerting to the user but the program will still function correctly.

Clikloops

If you add your own click loop and try to do something like update scroll bars you may run into trouble. Before your routine gets called, TextEdit will have set clipping down to just the `viewRect`. You will have to save away the old clipping region, set it out to sufficient size (–32767, –32767, 32767, 32767 is probably OK), do your drawing, then restore TextEdit's clipping area so that it can function properly.

#83: System Heap Size Warning

See also: The Memory Manager

Written by: Jim Friedlander June 21, 1986
Updated: March 1, 1988

Earlier versions of this note pointed out that, due to varying system heap sizes, the application heap does not always start at `$CB00`. The start of the application heap has not been fixed for some time now; programs that depend on it never work on the Macintosh SE or the Macintosh II.

#84: Edit File Format

Written by: Harvey Alcabes
Modified by: Bryan Johnson
Updated:

April 11, 1985
August 15, 1986
March 1, 1988

This technical note describes the format of the files created by Edit. It has been verified for versions 1.x and 2.0.

Edit, a text editor licensed by Apple and included in the Consulair 68000 Development System, can read any text-only file whose file type is TEXT. Files created by Edit have a creator ID of EDIT. Edit is a disk-based editor so the file length is not limited by available memory. Files created or modified by Edit, have the format described below; if they are not too long they can be read by any application which can read TEXT files (eg: MacWrite, Microsoft Word, or the APDA example program File).

The data fork contains text (ASCII characters). Carriage return characters indicate line breaks; tab characters are displayed as described below. No other characters have special significance.

The resource fork contains resources of type ETAB and EFNT. If Edit opens a text-only file that does not have these resources it will add them.

The ETAB (Editor TAB) resource, resource ID 1004, contains two integers. The first is the number of pixels to display for each space within a tab (not necessarily the same as for the space character). The second integer is the number of these spaces which will be displayed for each tab character.

The EFNT (Editor FoNT) resource, resource ID 1003, contains an integer followed by a Pascal string (length byte followed by characters). The integer is the point size of the document's font. The string contains the font name. If the string size (including the length byte) is odd, an extra byte is added so that the resource size is even.

For more information about Edit, contact:

Consulair Corp.
140 Campo Drive
Portola Valley, CA 94025
(415) 851-3272

#85: GetNextEvent; Blinking Apple Menu

See also: The Menu Manager
The Toolbox Event Manager
The Desk Manager

Written by: Rick Blair August 14, 1986
Updated: March 1, 1988

Wherein arcane mysteries are unraveled so you can make the Alarm Clock (or a similar desk accessory) blink the Apple menu at the appointed second. Also, why `GetNextEvent` is a good thing.

The obvious

Don't disable interrupts within an application! There will almost certainly come a time (or Macintosh) where you won't be able to change the interrupt mask because the processor is running in user mode. The one-second interrupt is used to blink the apple.

The not-so-obvious

You must call `GetNextEvent` periodically. `GetNextEvent` uses a filter (GNE filter) which allows for a routine to be installed which overrides (or augments) the behavior of the system. The GNE filter is installed by pointing the low-memory global `jGNEFilter` (a long word at `$29A`) to the routine. After all other GNE processing is complete, the `routine` will be called with `A1` pointing to the event record and `D0` containing the boolean result. The filter may then modify the event record or change the function result by altering the word on the stack at `4(A7)`. This word will match `D0` initially, of course.

A GNE filter is used to do the blinking when the interrupt handler has announced that the moment is at hand. `GetOSEvent` won't do. If you don't have a standard main event loop, it is generally a good idea to give `GetNextEvent` (and `SystemTask`, too) a call whenever you have any idle time. `GetNextEvent` "extra" services include, but aren't limited to, the following:

1. Calling the GNE filter.
2. Removing lingering disk-switched windows (uncommon unless memory is tight).
3. Making Window Manager activate, deactivate and update events happen.
4. Getting various events from a journaling driver when one is playing.
5. Giving `SystemEvent` a chance at each event.
6. Running command-shift function key routines (e.g. command-shift-4 to print the screen to an ImageWriter).

The more subtle

When the (default) GNE filter sees that the interrupt handler has set the "time to blink" flag, it looks at the first menu in `MenuList`. The title of that menu must consist solely of the "apple" character or no blinking will occur. It really just looks at the first word of the string to see if it is `$0114`. This is a Pascal string which has only the `$14` "apple" character in it. So you musn't have any spaces or any other characters in the title of your first menu or you'll get no blinkin' results.



#86: MacPaint Document Format

Revised by: Jim Reekes June 1989
Written by: Bill Atkinson 1983

This Technical Note describes the internal format of a MacPaint® document, which is a standard used by many other programs. This description is the same as that found in the "Macintosh Miscellaneous" section of early *Inside Macintosh* versions.
Changes since October 1988: Fixed bugs in the example code.

MacPaint documents are easy to read and write, and they have become a standard interchange format for full–page images on the Macintosh. This Note describes the MacPaint internal document format to help developers generate and interpret files in this format.

MacPaint documents have a file type of "PNTG," and since they use only the data fork, you can ignore the resource fork. The data fork contains a 512–byte header followed by compressed data which represents a single bitmap (576 pixels wide by 720 pixels tall). At a resolution of 72 pixels per inch, this bitmap occupies the full 8 inch by 10 inch printable area of a standard ImageWriter printer page.

Header

The first 512 bytes of the document form a header of the following format:

- 4–byte version number (default = 2)
- 38*8 = 304 bytes of patterns
- 204 unused bytes (reserved for future expansion)

As a Pascal record, the document format could look like the following:

```
MPHeader = RECORD
        Version:        LONGINT;
        PatArray:       ARRAY [1..38] of Pattern;
        Future:         PACKED ARRAY [1..204] of SignedByte;
END;
```

If the version number is zero, the document uses default patterns, so you can ignore the rest of the header block, and if your program generates MacPaint documents, you can write 512 bytes of zero for the document header. Most programs which read MacPaint documents can skip the header when reading.

Bitmap

Following the header are 720 compressed scan lines of data which form the 576 pixel wide by 720 pixel tall bitmap. Without compression, this bitmap would occupy 51,840 bytes and chew up disk space pretty fast; typical MacPaint documents compress to about 10K using the `_PackBits`

procedure to compress runs of equal bytes within each scan line. The bitmap part of a MacPaint document is simply the output of `_PackBits` called 720 times, with 72 bytes of input each time.

To determine the maximum size of a MacPaint file, it is worth noting what *Inside Macintosh* says about `_PackBits`:

> *"The worst case would be when `_PackBits` adds one byte to the row of bytes when packing."*

If we include an extra 512 bytes for the file header information to the size of an uncompressed bitmap (51,840), then the total number of bytes would be 52,352. If we take into account the extra 720 "potential" bytes (one for each row) to the previous total, the maximum size of a MacPaint file becomes 53,072 bytes.

Reading Sample

```
PROCEDURE ReadMPFile;
{ This is a small example procedure written in Pascal that demonstrates
  how to read MacPaint files. As a final step, it takes the data that
  was read and displays it on the screen to show that it worked.
  Caveat: This is not intended to be an example of good programming
  practice, in that the possible errors merely cause the program to exit.
  This is VERY uninformative, and there should be some sort of error handler
  to explain what happened. For simplicity, and thus clarity, those types
  of things were deliberately not included. This example will not work
  on a 128K Macintosh, since memory allocation is done too simplistically.
}

CONST
        DefaultVolume = 0;
        HeaderSize = 512;                   { size of MacPaint header in bytes }
        MaxUnPackedSize = 51840;            { maximum MacPaint size in bytes }
                                            { 720 lines * 72 bytes/line }

VAR
        srcPtr:         Ptr;
        dstPtr:         Ptr;
        saveDstPtr:     Ptr;
        lastDestPtr:    Ptr;
        srcFile:        INTEGER;
        srcSize:        LONGINT;
        errCode:        INTEGER;
        scanLine:       INTEGER;
        aPort:          GrafPort;
        theBitMap:      BitMap;

BEGIN
        errCode := FSOpen('MP TestFile', DefaultVolume, srcFile); { Open the file. }
        IF errCode <> noErr THEN ExitToShell;

        errcode := SetFPos(srcFile, fsFromStart, HeaderSize);     { Skip the header. }
        IF errCode <> noErr THEN ExitToShell;

        errCode := GetEOF(srcFile, srcSize);        { Find out how big the file is, }
        IF errCode <> noErr THEN ExitToShell;       { and figure out source size. }

        srcSize := srcSize - HeaderSize ;           { Remove the header from count. }
        srcPtr := NewPtr(srcSize);                  { Make buffer just the right size. }
        IF srcPtr = NIL THEN ExitToShell;

        errCode := FSRead(srcFile, srcSize, srcPtr); { Read the data into the buffer. }
        IF errCode <> noErr THEN ExitToShell;       { File marker is past header. }
```

```
        errCode := FSClose(srcFile);                { Close the file we just read. }
        IF errCode <> noErr THEN ExitToShell;

        { Create a buffer that will be used for the Destination BitMap. }
        dstPtr := NewPtrClear(MaxUnPackedSize);     {MPW library routine, see TN 219}
        IF dstPtr = NIL THEN ExitToShell;
        saveDstPtr := dstPtr;

        { Unpack each scan line into the buffer. Note that 720 scan lines are
          guaranteed to be in the file. (They may be blank lines.) In the
          UnPackBits call, the 72 is the count of bytes done when the file was
          created.  MacPaint does one scan line at a time when creating the file.
          The destination pointer is tested each time through the scan loop.
          UnPackBits should increment this pointer by 72, but in the case where
          the packed file is corrupted UnPackBits may end up sending bits into
          uncharted territory.  A temporary pointer "lastDstPtr" is used for testing
          the result.}

        FOR scanLine := 1 TO 720 DO BEGIN
             lastDstPtr := dstPtr;
             UnPackBits(srcPtr, dstPtr, 72);        { bumps both pointers }
             IF ORD4(lastDstPtr) + 72 <> ORD4(dstPtr) THEN ExitToShell;
        END;

        { The buffer has been fully unpacked. Create a port that we can draw into.
          You should save and restore the current port.  }
        OpenPort(@aPort);

        { Create a BitMap out of our saveDstPtr that can be copied to the screen. }
        theBitMap.baseAddr := saveDstPtr;
        theBitMap.rowBytes := 72;                   { width of MacPaint picture }
        SetPt(theBitMap.bounds.topLeft, 0, 0);
        SetPt(theBitMap.bounds.botRight, 72*8, 720); {maximum rectangle}

        { Now use that BitMap and draw the piece of it to the screen.
          Only draw the piece that is full screen size (portRect). }
        CopyBits(theBitMap, aPort.portBits, aPort.portRect,
                     aPort.portRect, srcCopy, NIL);

        { We need to dispose of the memory we've allocated.  You would not
          dispose of the destPtr if you wish to edit the data.  }
        DisposPtr(srcPtr);                  { dispose of the source buffer }
        DisposPtr(dstPtr);                  { dispose of the destination buffer }
END;
```

Writing Sample

```
PROCEDURE WriteMPFile;
{ This is a small example procedure written in Pascal that demonstrates how
  to write MacPaint files. It will use the screen as a handy BitMap to be
  written to a file.
}

CONST
        DefaultVolume = 0;
        HeaderSize = 512;                   { size of MacPaint header in bytes }
        MaxFileSize = 53072;                { maximum MacPaint file size. }

VAR
        srcPtr:         Ptr;
        dstPtr:         Ptr;
        dstFile:        INTEGER;
        dstSize:        LONGINT;
        errCode:        INTEGER;
        scanLine:       INTEGER;
        aPort:          GrafPort;
        dstBuffer:      PACKED ARRAY[1..HeaderSize] OF BYTE;
        I:              LONGINT;
        picturePtr:     Ptr;
        tempPtr:        BigPtr;
        theBitMap:      BitMap;

BEGIN
        { Make an empty buffer that is the picture size. }
        picturePtr := NewPtrClear(MaxFileSize);     {MPW library routine, see TN 219}
        IF picturePtr = NIL THEN ExitToShell;

        { Open a port so we can get to the screen's BitMap easily.  You should save
          and restore the current port. }
        OpenPort(@aPort);

        { Create a BitMap out of our dstPtr that can be copied to the screen. }
        theBitMap.baseAddr := picturePtr;
        theBitMap.rowBytes := 72;                   { width of MacPaint picture }
        SetPt(theBitMap.bounds.topLeft, 0, 0);
        SetPt(theBitMap.bounds.botRight, 72*8, 720); {maximum rectangle}

        { Draw the screen over into our picture buffer. }
        CopyBits(aPort.portBits, theBitMap, aPort.portRect,
                    aPort.portRect, srcCopy, NIL);

        { Create the file, giving it the right Creator and File type.}
        errCode := Create('MP TestFile', DefaultVolume, 'MPNT', 'PNTG');
        IF errCode <> noErr THEN ExitToShell;

        { Open the data file to be written. }
        errCode := FSOpen(dstFileName, DefaultVolume, dstFile);
        IF errCode <> noErr THEN ExitToShell;

        FOR I := 1 to HeaderSize DO                 { Write the header as all zeros. }
             dstBuffer[I] := 0;
        errCode := FSWrite(dstFile, HeaderSize, @dstBuffer);
        IF errCode <> noErr THEN ExitToShell;
```

```
        { Now go into a loop where we pack each line of data into the buffer,
          then write that data to the file. We are using the line count of 72
          in order to make the file readable by MacPaint. Note that the
          Pack/UnPackBits can be used for other purposes. }
        srcPtr := theBitMap.baseAddr;              { point at our picture BitMap }
        FOR scanLine := 1 to 720 DO
            BEGIN
               dstPtr := @dstBuffer;               { reset the pointer to bottom }
               PackBits(srcPtr, dstPtr, 72);              { bumps both ptrs }
               dstSize := ORD(dstPtr)-ORD(@dstBuffer);    { calc packed size }
               errCode := FSWrite(dstFile, dstSize, @dstBuffer);
               IF errCode <> noErr THEN ExitToShell;
            END;

        errCode := FSClose(dstFile);               { Close the file we just wrote. }
        IF errCode <> noErr THEN ExitToShell;
END;
```

Further Reference:

- *Inside Macintosh*, Volume I-135, QuickDraw
- *Inside Macintosh*, Volume I-465, Toolbox Utilities
- *Inside Macintosh*, Volume II-77, The File Manager
- Technical Note #219, New Memory Manager Glue Routines

MacPaint is a registered trademark of Claris Corporation.

#87: Error in FCBPBRec

See also: The File Manager

Written by: Jim Friedlander August 18, 1986
Updated: March 1, 1988

The declaration of a `FCBPBRec` is wrong in *Inside Macintosh Volume IV* and early versions of MPW. This has been fixed in MPW 1.0 and newer.

An error was made in the declaration of an `FCBPBRec` parameter block that is used in `PBGetFCBInfo` calls. The field `ioFCBIndx` was incorrectly listed as a `LONGINT`. The following declaration (found in *Inside Macintosh*):

```
...
ioRefNum:       INTEGER;
filler:         INTEGER;
ioFCBIndx:      LONGINT;
ioFCBFlNm:      LONGINT;
...
```

should be changed to:

```
...
ioRefNum:       INTEGER;
filler:         INTEGER;
ioFCBIndx:      INTEGER;
ioFCBFiller1:   INTEGER;
ioFCBFlNm:      LONGINT;
...
```

Macintosh Technical Notes



#88: Signals

See also: Using Assembly Language (Mixing Pascal & Assembly)

Written by: Rick Blair August 1, 1986
Updated: March 1, 1988

Signals are a form of intra-program interrupt which can greatly aid clean, inexpensive error trapping in stack frame intensive languages. A program may invoke the `Signal` procedure and immediately return to the last invocation of `CatchSignal`, including the complete stack frame state at that point.

Signals allow a program to leave off execution at one point and return control to a convenient error trap location, regardless of how many levels of procedure nesting are in between.

The example is provided with a Pascal interface, but it is easily adapted to other languages. The only qualification is that the language **must** bracket its procedures (or functions) with `LINK` and `UNLK` instructions. This will allow the signal code to clean up at procedure exit time by removing `CatchSignal` entries from its internal queue. **Note:** only procedures and/or functions that call `CatchSignal` need to be bracketed with `LINK` and `UNLK` instructions.

Important: `InitSignals` must be called from the main program so that `A6` can be set up properly.

Note that there is no limit to the number of local `CatchSignals` which may occur within a single routine. Only the last one executed will apply, of course, unless you call `FreeSignal`. `FreeSignal` will "pop" off the last `CatchSignal`. If you attempt to `Signal` with no `CatchSignals` pending, `Signal` will halt the program with a debugger trap.

`InitSignals` creates a small relocatable block in the application heap to hold the signal queue. If `CatchSignal` is unable to expand this block (which it does 5 elements at a time), then it will signal back to the last successful `CatchSignal` with `code` = 200. A `Signal(0)` acts as a `NOP`, so you may pass `OSErrs`, for instance, after making File System type calls, and, if the `OSErr` is equal to `NoErr`, nothing will happen.

`CatchSignal` may not be used in an expression if the stack is used to evaluate that expression. For example, you can't write:

```
c:= 3*CatchSignal;
```

"Gotcha" summary

1. Routines which call `CatchSignal` must have stack frames.
2. `InitSignals` must be called from the outermost (main) level.
3. Don't put the `CatchSignal` function in an expression. Assign the result to an `INTEGER variable; i.e. i:=CatchSignal.`
4. It's safest to call a procedure to do the processing after `CatchSignal` returns. See the Pascal example `TestSignals` below. This will prevent the use of a variable which may be held in a register.

Below are three separate source files. First is the Pascal interface to the signaling unit, then the assembly language which implements it in MPW Assembler format. Finally, there is an example program which demonstrates the use of the routines in the unit.

```
{File ErrSignal.p}
UNIT ErrSignal;

INTERFACE

{Call this right after your other initializations (InitGraf, etc.)--in other
words as early as you can in the application}
PROCEDURE InitSignals;

{Until the procedure which encloses this call returns, it will catch
subsequent Signal calls, returning the code passed to Signal. When
CatchSignal is encountered initially, it returns a code of zero. These calls
may "nest"; i.e. you may have multiple CatchSignals in one procedure.
Each nested CatchSignal call uses 12 bytes of heap space }
FUNCTION CatchSignal:INTEGER;

{This undoes the effect of the last CatchSignal. A Signal will then invoke
the CatchSignal prior to the last one.}
PROCEDURE FreeSignal;

{Returns control to the point of the last CatchSignal. The program will then
behave as though that CatchSignal had returned with the code parameter
supplied to Signal.}
PROCEDURE Signal(code:INTEGER);

END.
{End of ErrSignal.p}
```

Here's the assembly source for the routines themselves:

```
; ErrSignal code w. InitSignal, CatchSignal,FreeSignal, Signal
; defined
;
;               Version 1.0 by Rick Blair

            PRINT     OFF
            INCLUDE   'Traps.a'
            INCLUDE   'ToolEqu.a'
            INCLUDE   'QuickEqu.a'
            INCLUDE   'SysEqu.a'
            PRINT     ON


CatchSigErr EQU       200             ;"insufficient heap" message
SigChunks   EQU       5               ;number of elements to expand by
FrameRet    EQU       4               ;return addr. for frame (off A6)
SigBigA6    EQU       $FFFFFFFF       ;maximum positive A6 value


; A template in MPW Assembler describes the layout of a collection of data
; without actually allocating any memory space. A template definition starts ;
with a RECORD directive and ends with an ENDR directive.

; To illustrate how the template type feature works, the following template
; is declared and used. By using this, the assembler source appromixates very
; closely Pascal source for referencing the corresponding information.

;template for our table elements
  SigElement RECORD    0     ;the zero is the template origin
  SigSP      DS.L      1     ;the SP at the CatchSignal—(DS.L just like EQU)
  SigRetAddr DS.L      1     ;the address where the CatchSignal returned
  SigFRet    DS.L      1     ;return addr. for encl. procedure
  SigElSize  EQU       *     ;just like EQU 12
             ENDR


; The global data used by these routines follows. It is in the form of a
; RECORD, but, unlike above, no origin is specified, which means that memory
; space *will* be allocated.
; This data is referenced through a WITH statement at the beginning of the
; procs that need to get at this data. Since the Assembler knows when it is
; referencing data in a data module (since they must be declared before they
; are accessed), and since such data can only be accessed based on A5, there
; is no need to explicitly specify A5 in any code which references the data
; (unless indexing is used). Thus, in this program we have omitted all A5
; references when referencing the data.

  SigGlobals RECORD                  ;no origin means this is a data record
                                     ;not a template(as above)
  SigEnd     DS.L      1             ;current end of table
  SigNow     DS.L      1             ;the MRU element
  SigHandle  DC.L      0             ;handle to the table
             ENDR
```

```
InitSignals PROC     EXPORT          ;PROCEDURE InitSignals;

            IMPORT   CatchSignal
            WITH     SigElement,SigGlobals

;the above statement makes the template SigElement and the global data
;record SigGlobals available to this procedure

            MOVE.L   #SigChunks*SigElSize,D0
            _NewHandle               ;try to get a table
            BNE.S    forgetit        ;we couldn't get that!?

            MOVE.L   A0,SigHandle    ;save it
            MOVE.L   #-SigElSize,SigNow ;point "now" before start
            MOVE.L   #SigChunks*SigElSize,SigEnd ;save the end
            MOVE.L   #SigBigA6,A6    ;make A6 valid for Signal
forgetit    RTS
            ENDP

CatchSignal PROC     EXPORT          ;FUNCTION CatchSignal:INTEGER;
            IMPORT   SiggySetup,Signal,SigDeath
            WITH     SigElement,SigGlobals

            MOVE.L   (SP)+,A1        ;grab return address
            MOVE.L   SigHandle,D0    ;handle to table
            BEQ      SigDeath        ;if NIL then croak
            MOVE.L   D0,A0           ;put handle in A-register
            MOVE.L   SigNow,D0
            ADD.L    #SigElSize,D0
            MOVE.L   D0,SigNow       ;save new position
            CMP.L    SigEnd,D0       ;have we reached the end?
            BNE.S    catchit         ;no, proceed

            ADD.L    #SigChunks*SigElSize,D0 ;we'll try to expand
            MOVE.L   D0,SigEnd       ;save new (potential) end
            _SetHandleSize
            BEQ.S    @0              ;jump around if it worked!

;signals, we use 'em ourselves
            MOVE.L   SigNow,SigEnd   ;restore old ending offset
            MOVE.L   #SigElSize,D0
            SUB.L    D0,SigNow       ;ditto for current position
            MOVE.W   #catchSigErr,(SP);we'll signal a "couldn't
                                     ;                 catch" error
            JSR      Signal          ;never returns of course


@0          MOVE.L   SigNow,D0

catchit     MOVE.L   (A0),A0         ;deref.
            ADD.L    D0,A0           ;point to new entry
            MOVE.L   SP,SigSP(A0)    ;save SP in entry
            MOVE.L   A1,SigRetAddr(A0) ;save return address there
            CMP.L    #SigBigA6,A6    ;are we at the outer level?
            BEQ.S    @0              ;yes, no frame or cleanup needed
            MOVE.L   FrameRet(A6),SigFRet(A0);save old frame return
                                     ;                address
```

```
            LEA       SiggyPop,A0
            MOVE.L    A0,FrameRet(A6) ;set cleanup code address
@0          CLR.W     (SP)            ;no error code (before its time)
            JMP       (A1)            ;done setting the trap

SiggyPop    JSR       SiggySetup      ;get pointer to element
            MOVE.L    SigFRet(A0),A0 ;get proc's real return address
            SUB.L     #SigElSize,D0
            MOVE.L    D0,SigNow       ;"pop" the entry
            JMP       (A0)            ;gone
            ENDP

FreeSignal  PROC      EXPORT          ;PROCEDURE FreeSignal;
            IMPORT    SiggySetup
            WITH      SigElement,SigGlobals
            JSR       SiggySetup      ;get pointer to current entry
            MOVE.L    SigFRet(A0),FrameRet(A6) ;"pop" cleanup code
            SUB.L     #SigElSize,D0
            MOVE.L    D0,SigNow       ;"pop" the entry
            RTS
            ENDP

Signal      PROC      EXPORT          ;PROCEDURE Signal(code:INTEGER);
            EXPORT    SiggySetup,SigDeath
            WITH      SigElement,SigGlobals
            MOVE.W    4(SP),D1        ;get code
            BNE.S     @0              ;process the signal if code is non-zero
            MOVE.L    (SP),A0         ;save return address
            ADDQ.L    #6,SP           ;adjust stack pointer
            JMP       (A0)            ;return to caller(code was 0)

@0          JSR       SiggySetup      ;get pointer to entry
            BRA.S     SigLoop1

SigLoop     UNLK      A6              ;unlink stack by one frame
SigLoop1    CMP.L     SigSP(A0),A6    ;is A6 beyond the saved stack?
            BLO.S     SigLoop         ;yes, keep unlinking
            MOVE.L    SigSP(A0),SP    ;bring back our SP
            MOVE.L    SigRetAddr(A0),A0 ;get return address
            MOVE.W    D1,(SP)         ;return code to CatchSignal
            JMP       (A0)            ;Houston, boost the Signal!
                      ;(or Hooston if you're from the Negative Zone)

SiggySetup  MOVE.L    SigHandle,A0
            MOVE.L    (A0),A0         ;deref.
            MOVE.L    A0,D0           ;to set CCR
            BEQ.S     SigDeath        ;nil handle means trouble
            MOVE.L    SigNow,D0       ;grab table offset to entry
            BMI.S     SigDeath        ;if no entries then give up
            ADD.L     D0,A0           ;point to current element
            RTS

SigDeath    _Debugger                 ;a signal sans catch is bad news

            ENDP
            END
```

Now for the example Pascal program:

```
PROGRAM TestSignals;
USES ErrSignal;

VAR i:INTEGER;

PROCEDURE DoCatch(s:STR255; code:INTEGER);
BEGIN
  IF code<>0 THEN BEGIN
    Writeln(s,code);
    Exit(TestSignals);
  END;
END; {DoCatch}

PROCEDURE Easy;
  PROCEDURE Never;
    PROCEDURE DoCatch(s:STR255; code:INTEGER);
    BEGIN
      IF code<>0 THEN BEGIN
        Writeln(s,code);
        Exit(Never);
      END;
    END; {DoCatch}

  BEGIN {Never}
  i:=CatchSignal;
  DoCatch('Signal caught from Never, code = ', i );

  i:=CatchSignal;
  IF i<>0 THEN DoCatch('Should never get here!',i);

  FreeSignal; {"free" the last CatchSignal}
  Signal(7); {Signal a 7 to the last CatchSignal}
  END;{Never}
BEGIN {Easy}
Never;
Signal(69);        {this won't be caught in Never}
END;{Easy}         {all local CatchSignals are freed when a procedure exits.}

BEGIN {PROGRAM}
InitSignals; {You must call this early on!}

{catch Signals not otherwise caught by the program}
i:=CatchSignal;
IF i<>0 THEN
 DoCatch('Signal caught from main, code = ',i);

Easy;
END.
```

The example program produces the following two lines of output:

```
Signal caught from Never, code = 7
Signal caught from main, code = 69
```

#89: DrawPicture Bug

Written by: Ginger Jernigan August 16, 1986
Updated: March 1, 1988

Earlier versions of this note described a bug in `DrawPicture`. This bug never occurred on 64K ROM machines, and has been fixed in System 3.2 and newer. Use of Systems older than 3.2 on non-64K ROM machines is no longer recommended.

#90: SANE Incompatibilities

Written by: Mark Baumwell August 14, 1986
Updated: March 1, 1988

Earlier versions of this note described a problem with SANE and System 2.0. Use of System 2.0 is only recommended for Macintosh 128 machines, which contain the 64K ROMs. Information specific to 64K ROM machines has been deleted from Macintosh Technical Notes for reasons of clarity.

#91: Optimizing for the LaserWriter—Picture Comments

See also:
- The Print Manager
- QuickDraw
- Technical Note #72—Optimizing for the LaserWriter—Techniques
- Technical Note #27—MacDraw Picture Comments
- *PostScript Language Reference Manual*, Adobe Systems
- *PostScript Language Tutorial and Cookbook*, Adobe Systems
- *LaserWriter Reference Manual*

Written by:	Ginger Jernigan	November 15, 1986
Modified by:	Ginger Jernigan	March 2, 1987
Updated:		March 1, 1988

This technical note is a continuation of Technical Note #72. This technical note discusses the picture comments that the LaserWriter driver recognizes.

This technical note has been modified to include corrected descriptions of the `SetLineWidth`, `PostScriptFile` and `ResourcePS` comments and to include some additional warnings.

The implementation of QuickDraw's `picComment` facility by the LaserWriter driver allows you to take advantage of features (like rotated text) which are available in PostScript but may not be available in QuickDraw.

Warning: Using PostScript-specific comments will make your code printer-dependent and may cause compatibility problems with non-PostScript devices, so don't use them unless you absolutely have to.

Some of the picture comments below are designed to be issued along with QuickDraw commands that simulate the commented commands on the Macintosh screen. When the comments are used, the accompanying QuickDraw comments are ignored. If you are designing a picture to be printed by the LaserWriter, the structure and use of these comments must be precise, otherwise nothing will print. If another printer driver (like the ImageWriter I/II driver) has not implemented these comments, the comments are ignored and the accompanying QuickDraw commands are used.

Below are the picture comments that the LaserWriter driver recognizes:

Type	Kind	Data Size	Data	Description
TextBegin	150	6	TTxtPicRec	Begin text function
TextEnd	151	0	NIL	End text function
StringBegin	152	0	NIL	Begin pieces of original string
StringEnd	153	0	NIL	End pieces of original string
TextCenter	154	8	TTxtCenter	Offset to center of rotation
* LineLayoutOff	155	0	NIL	Turns LaserWriter line layout off
* LineLayoutOn	156	0	NIL	Turns LaserWriter line layout on
PolyBegin	160	0	NIL	Begin special polygon
PolyEnd	161	0	NIL	End special polygon
PolyIgnore	163	0	NIL	Ignore following poly data
PolySmooth	164	1	PolyVerb	Close, Fill, Frame
picPlyClo	165	0	NIL	Close the poly
* DashedLine	180	-	TDashedLine	Draw following lines as dashed
* DashedStop	181	0	NIL	End dashed lines
* SetLineWidth	182	4	Point	Set fractional line widths
* PostScriptBegin	190	0	NIL	Set driver state to PostScript
* PostScriptEnd	191	0	NIL	Restore QuickDraw state
* PostScriptHandle	192	-	PSData	PostScript data in handle
*† PostScriptFile	193	-	FileName	FileName in data handle
* TextIsPostScript	194	0	NIL	QuickDraw text is sent as PostScript
*† ResourcePS	195	8	Type/ID/Index	PostScript data in a resource file
**RotateBegin	200	4	TRotation	Begin rotated port
**RotateEnd	201	0	NIL	End rotation
**RotateCenter	202	8	Center	Offset to center of rotation
**FormsPrinting	210	0	NIL	Don't clear print buffer after each page
**EndFormsPrinting	211	0	NIL	End forms printing after PrClosePage

* These comments are only implemented in LaserWriter driver 3.0 or later.
** These comments are only implemented in LaserWriter driver 3.1 or later.
† These comments are not available when background printing is enabled.

Each of these comments are discussed below in six groups: Text, Polygons, Lines, PostScript, Rotation, and Forms. Code examples are given where appropriate. For other examples of how to use picture comments for printing please see the Print example program in the Software Supplement (currently available through APDA as "Macintosh Example Applications and Sources 1.0").

Note: The examples used in the *LaserWriter Reference Manual* are incorrect. Please use the examples presented here instead.

Text

In order to support the What-You-See-Is-What-You-Get paradigm, the LaserWriter driver uses a line layout algorithm to assure that the placement of the line on the printer closely approximates the placement of the line on the screen. This means that the printer driver gets the width of the line from QuickDraw, then tells PostScript to place the text in exactly the same place with the same width.

The `TextBegin` comment allows the application to specify the layout and the orientation of the text that follows it by specifying the following information:

```
TTxtPicRec = PACKED RECORD
  tJus:  Byte;     {0,1,2,3,4 or greater => none, left, center, right, full
                   justification }
  tFlip: Byte;     {0,1,2 => none, horizontal, vertical coordinate flip }
  tRot:  INTEGER;  {0..360 => clockwise rotation in degrees }
  tLine: Byte;     {1,2,3.. => single, 1-1/2, double.. spacing }
  tCmnt: Byte;     {Reserved }
END; { TTxtPicRec }
```

Left, right or center justification, specified by `tJust`, tells the driver to maintain only the left, right or center point, without recalculating the interword spacing. Full justification specifies that both endpoints be maintained and interword spacing be recalculated. This means that the driver makes sure that the specified points are maintained on the printer without caring whether the overall width has changed. Full justification means that the overall width of the line has been maintained. `tFlip` and `tRot` specify the orientation of the text, allowing the application to take advantage of the rotation features of PostScript. `tLine` specifies the interline spacing. When no `TextBegin` comment is used, the defaults are full justification, no rotation and single-spaced lines.

String Reconstruction

The `StringBegin` and `StringEnd` comments are used to bracket short strings of text that are actually sections of an original long string. MacDraw, for instance, breaks long strings into shorter pieces to avoid stack overflow problems with QuickDraw in the 64K ROM. When these smaller strings are bracketed by `StringBegin` and `StringEnd`, the LaserWriter driver assumes that the enclosed strings are parts of one long string and will perform its line layout accordingly. Erasing or filling of background rectangles should take place before the `StringBegin` comment to avoid confusing the process of putting the smaller strings back together.

Text Rotation

In order to rotate a text object, PostScript needs to have information concerning the center of rotation. The `TextCenter` comment provides this information when a rotation is specified in the `TextBegin` comment. This comment contains the offset from the present pen location to the center of rotation. The offset is given as the y-component, then the x-component, which are declared as fixed-point numbers. This allows the center to be in the middle of a pixel. This comment should appear after the `TextBegin` comment and before the first following `StringBegin` comment.

The associated comment data looks like this:

```
TTxtCenter = RECORD
   y,x: Fixed;       {offset from current pen location to center of rotation}
END; { TTxtCenter }
```

Right after a `TextBegin` comment, the LaserWriter driver expects to see a `TextCenter` comment specifying the center of rotation for any text enclosed within the text comment calls. It will ignore all further `CopyBits` calls, and print all standard text calls in the rotation specified by the information in `TTxtPicRec`. The center of rotation is the offset from the beginning position of the first string following the `TextCenter` comment. The printer driver also expects the string locations to be in the coordinate system of the current QuickDraw port. The printer driver rotates the entire port to draw the text so it can draw several strings with one rotation comment and one center comment. It is good practice to enclose an entire paragraph or paragraphs of text in a single rotation comment so that the driver makes the fewest number of rotations.

The printer driver can draw non-textual objects within the bounds of the text rotation comments but it must unrotate to draw the object, then re-rotate to draw the next string of text. To do this the printer driver must receive another `TextCenter` comment before each new rotation. So, rotated text and unrotated objects can be drawn inter-mixed within one `TextBegin`/`TextEnd` comment pair, but performance is slowed.

Note that all bit maps and all clip regions are ignored during text rotation so that clip regions can be used to clip out the strings on printers that can't take advantage of these comments. This has the unfortunate side effect of not allowing rotated text to be clipped.

Rotated text comments are not associated with landscape and portrait orientation of the printer paper as selected by the Page Setup dialog. These are rotations with reference to the current QuickDraw port only.

All of the above text comments are terminated by a `TextEnd` comment.

Turning Off Line Layout

If your application is using its own line layout algorithm (it uses its own character widths or does its own character or word placement), the printer driver doesn't need to do it too. To turn off line layout, you can use the `LineLayoutOff` comment. `LineLayoutOn` turns it on again.

Turning on `FractEnable` for the 128K ROMs has the same effect as `LineLayoutOff`. When the driver detects that `FractEnable` has been turned on, line layout is not performed. The driver assumes that all text being printed is already spaced correctly for the LaserWriter and just sends it as is.

Polygons

The polygon comments are recognized by the LaserWriter driver because they are used by MacDraw as an alternate method of defining polygons.

The `PolyBegin` and `PolyEnd` comments bracket polygon line segments, giving an alternate way to specify a polygon. All `StdLine` calls between these two comments are part of the polygon. The endpoints of the lines are the vertices of the polygon.

The `picPlyClo` comment specifies that the current polygon should be closed. This comes immediately after `PolyBegin`, if at all. It is not sufficient to simply check for `begPt = endPt`, since MacDraw allows you to create a "closed" polygon that isn't really closed. This comment is especially critical for smooth curves because it can make the difference between having a sharp corner or not in the curve.

These comments also work with the `StdPoly` call. If a `FillRgn` is encountered before the `PolyEnd` comment, then the polygon is filled. Unlike QuickDraw polygons, comment polygons do not require an initial `MoveTo` call within the scope of the polygon comment. The polygon will be drawn using the current pen location at the time the polygon comment is received. The pen must be set before the polygon comment is called.

Splines

A spline is a method used to determine the smallest number of points that define a curve. In MacDraw, splines are used as a method for smoothing polygons. The vertices of the underlying unsmoothed polygon are the control nodes for the quadratic B-spline curve which is drawn. PostScript has a direct facility for cubic B-splines and the LaserWriter translates the quadratic B-spline nodes it gets into the appropriate nodes for a cubic B-spline that will exactly emulate the original quadratic B-spline.

The `PolySmooth` comment specifies that the current polygon should be smoothed. This comment also contains data that provides a means of specifying which verbs to use on the smoothed polygon (bits 7 through 3 are not currently assigned):

```
TPolyVerb = PACKED RECORD
   f7, f6, f5, f4, f3, fPolyClose, fPolyFill, fPolyframe : Boolean;
END; { TPolyVerb }
```

Although the closing information is redundant with the `picPlyClo` comment, it is included for the convenience of the LaserWriter.

The LaserWriter uses the pen size at the time the `PolyBegin` comment is received to frame the smoothed polygon if framing is called for by the `TPolyVerb` information. When the `PolyIgnore` comment is received by the LaserWriter driver, all further `StdLine` calls are ignored until the `PolyEnd` comment is encountered. For polygons that are to be smoothed, set the initial pen width to zero after the `PolyBegin` comment so that the unsmoothed polygon will not be drawn by other printers not equipped to handle polygon comments. To fill the polygon, call `StdRgn` with the fill verb and the appropriate pattern set, as well as specifying fill in the `PolySmooth` comment.

Lines

The `DashedLine` and `DashedLineStop` comments are used to communicate PostScript information for drawing dashed lines.

The `DashedLine` comment contains the following additional data:

```
TDashedLine = PACKED RECORD
  offset:   SignedByte;                {Offset as specified by PostScript}
  centered: SignedByte;                {Whether dashed line should be
                                        centered to begin and end points}
  dashed:   Array[0..1] of SignedByte; {1st byte is # bytes following}
END; { TDashedLine }
```

The printer driver sets up the PostScript dashed line command, as defined on page 214 of Adobe's *PostScript Language Reference Manual*, using the parameters specified in the comment. You can specify that the dashed line be centered between the begin and end points of the lines by making the `centered` field nonzero.

The `SetLineWidth` comment allows you to set the pen width of all subsequent objects drawn. The additional data is a point. The vertical portion of the point is the numerator and the horizontal portion is the denominator of the scaling factor that the horizontal and vertical components of the pen are then multiplied by to obtain the new pen width. For example, if you have a pen size of 1,2 and in your line width comment you use 2 for the horizontal of the point and 7 for the vertical, the pen size will then be (7/2)*1 pixels wide and (7/2)*2 pixels high.

Below is an example of how to use the line comments:

```
PROCEDURE LineTest;
{This procedure shows how to do dashed lines and how to change the line width}
CONST
  DashedLine = 180;
  DashedStop = 181;
  SetLineWidth = 182;

TYPE
  DashedHdl = ^DashedPtr;
  DashedPtr = ^TDashedLine;
  TDashedLine = PACKED RECORD
    offset: SignedByte;
    Centered: SignedByte;
    dashed: Array[0..1] of SignedByte;    { the 0th element is the length }
  END; { TDashedLine }
  widhdl = ^widptr;
  widptr = ^widpt;
  widpt = Point;


VAR
  arect    : rect;
  Width    : Widhdl;
  dashedln : DashedHdl;
```

```
BEGIN {LineTest}
  Dashedln := dashedhdl(NewHandle(sizeof(tdashedline)));
  Dashedln^^.offset := 0;        { No offset}
  Dashedln^^.centered := 0;      { don't center}
  Dashedln^^.dashed[0] := 1;     { this is the length }
  Dashedln^^.dashed[1] := 8;     { this means 8 points on, 8 points off }

  Width := widhdl(NewHandle(sizeof(widpt)));
  Width^^.h := 2;                { denominator is 2}
  Width^^.v := 7;                { numerator is 7}

  myPic := OpenPicture(theWorld);
    SetPen(1,2);                 { Set the pen size to 1 wide x 2 high }
    ClipRect(theWorld);
    MoveTo(20,20);
    DrawString('Do line test');
    PicComment(DashedLine,GetHandleSize(Handle(dashedln)),Handle(dashedln));
    PicComment(SetLineWidth,4,Handle(width));    {SetLineWidth}
    SetRect(arect,100,100,500,500);
    FrameRect(aRect);
    MoveTo(500,500);
    Lineto(100,100);
    PicComment(DashedStop,0,nil);                {DashedStop}
  ClosePicture;
  DisposHandle(handle(width));                   {Clean up}
  DisposHandle(handle(dashedln));
  PrintThePicture;                               {print it please}
  KillPicture(MyPic);
END; {LineTest}
```

PostScript

The PostScript comments tell the printer driver that the application is going to be communicating with the LaserWriter directly using PostScript commands instead of QuickDraw. The driver sends the accompanying PostScript to the printer with no preprocessing and no error checking. The application can specify data in the comment handle itself or point to another file which contains text to send to the printer. When the application is finished sending PostScript, the `PostScriptEnd` comment tells the printer driver to resume normal QuickDraw mode.

Any Quickdraw drawing commands made by the application between the `PostScriptBegin` and `PostScriptEnd` comments will be ignored by PostScript printers. In order to use PostScript in a device independent way, you should always include two representations of your document. The first representation should be a series of Quickdraw drawing commands. The second representation of your document should be a series of PostScript commands, sent to the Printing Manager via picture comments. This way, when you are printing to a PostScript device, the picture comments will be executed, and the Quickdraw commands ignored. When printing to a non-PostScript device, the picture comments will be ignored, and the Quickdraw commands will be executed. This method allows you to use PostScript, without having to ask the device if it supports it. This allows your application to get the best results with any printer, without being device dependent.

Here are some guidelines you need to remember:

- The graphic state set up during QuickDraw calls is maintained and is not affected by PostScript calls made with these comments.

- The header has changed a number of parameters so sometimes you won't get the results you expect. You may want to take a look at the header listed in *The LaserWriter Reference Manual* available through APDA.

- The header changes the PostScript coordinate system so that the origin is at the top-left corner of the page instead of at the bottom-left corner. This is done so that the QuickDraw coordinates that are used don't have to be remapped into the standard PostScript coordinate system. If you don't allow for this, all drawing is printed upside down. Please see the *PostScript Language Reference Manual* for details about transformation matrices.

- Don't call `showpage`. This is done for you by the driver. If you do, you won't be able to switch back to QuickDraw mode and an additional page will be printed when you call `PrClosePage`.
- Don't call `exitserver`. You may get very strange results.
- Don't call `initgraphics`. Graphics states are already set up by the header.

- Don't do anything that you expect to live across jobs.

- You won't be able to interrogate the printer to get information back through the driver.

The PostScriptBegin comment sets the driver state to prepare for the generation of PostScript by the application by calling gsave to save the current state. PostScript is then sent to the printer by using comments 192 through 195. The QuickDraw state of the driver is then restored by the PostScriptEnd comment. All QuickDraw operations that occur outside of these comments are performed; no clipping occurs as with the text rotation comments.

PostScript From a Text Handle

When the PostScriptHandle comment is used, the handle PSData points to the PostScript commands which are sent. PSData is a generic handle that points to text, without a length byte. The text is terminated by a carriage return. This comment is terminated by a PostScriptEnd comment.

Note: Due to a bug in the 3.1 LaserWriter driver, PostScriptEnd will not restore the QuickDraw state after the use of a PostScriptHandle comment. The workaround is to only use this comment at the end of your drawing, after you have made all the QuickDraw calls you need. This problem is fixed in more recent versions of the driver.

Here's an example of how to use this comment:

```
PROCEDURE PostHdl;
{this procedure shows how to use PostScript from a text Handle}
CONST
  PostScriptBegin = 190;
  PostScriptEnd = 191;
  PostScriptHandle = 192;

VAR
  MyString  : Str255;
  tempstr   : String[1];
  MyHandle  : Handle;
  err       : OSErr;

BEGIN { PostHdl }
  MyString := '/Times-Roman findfont 12 scalefont setfont 230 600 moveto
              (Hello World) show';
  tempstr:=' ';
  tempstr[1] := chr(13); {has to be terminated by a carriage return }
  MyString := Concat(MyString, tempstr); { in order for it to execute}
  err := PtrToHand (Pointer(ord(@myString)+1), MyHandle, length(MyString));
  MyPic := OpenPicture(theWorld);
    ClipRect(theWorld);
    MoveTo(20,20);
    DrawString('PostScript from a Handle');
    PicComment(PostScriptBegin,0,nil);      {Begin PostScript}
    PicComment(PostScriptHandle,length(mystring),MyHandle);
    PicComment(PostScriptEnd,0,nil);        {PostScript End}
  ClosePicture;
  DisposHandle(MyHandle);                   {Clean up}
  PrintThePicture;                          {print it please}
  KillPicture(MyPic);
END; { PostHdl }
```

Defining PostScript as QuickDraw Text

All QuickDraw text following the `TextIsPostScript` comment is sent as PostScript. No error checking is performed. This comment is terminated by a `PostScriptEnd` comment.

Here is an example:

```
PROCEDURE PostText;
{Shows how to use PostScript in strings in a QuickDraw picture}
CONST
  PostScriptBegin = 190;
  PostScriptEnd = 191;
  TextIsPostScript = 194;

BEGIN { PostTest }
  MyPic := OpenPicture(theWorld);
    ClipRect(theWorld);
    MoveTo(20,20);
    DrawString('TextIsPostScript Comment');
    PicComment(PostScriptBegin,0,nil);      {Begin PostScript}
    PicComment(TextIsPostScript,0,nil);     {following text is PostScript}
      DrawString('0 728 translate');        {move the origin and rotate the}
      DrawString('1 -1 scale');             {coordinate system}

      DrawString('newpath');
      DrawString('100 470 moveto');
      DrawString('500 470 lineto');
      DrawString('100 330 moveto');
      DrawString('500 330 lineto');
      DrawString('230 600 moveto');
      DrawString('230 200 lineto');
      DrawString('370 600 moveto');
      DrawString('370 200 lineto');
      DrawString('10 setlinewidth');
      DrawString('stroke');
      DrawString('/Times-Roman findfont 12 scalefont setfont');
      DrawString('230 600 moveto');
      DrawString('(Hello World) show');
    PicComment(PostScriptEnd,0,nil);        {PostScriptEnd}
  ClosePicture;
  PrintThePicture;                                  {print it please}
  KillPicture(MyPic);
END; { PostText }
```

PostScript From a File

The `PostScriptFile` and `ResourcePS` comments allow you to send PostScript to the printer from a resource file. Before these comments are described there are some restrictions you need to follow:

- Don't ever copy a picture containing these comments to the clipboard. If it is pasted into another application and the specified file or resource is not available, printing will be aborted and the user won't know what went wrong. This could be very confusing to a user. If you want the PostScript information to be available when printed from another application, use one of the other comments and include the information in the picture.
- Don't keep the PostScript in a separate file from the actual data file. If the data file ever gets moved without the PostScript file, when the picture is printed the data file may not be found and the print job will be aborted, again without the user knowing what went wrong. Keeping the data and PostScript in the same file will forestall many headaches for you and the user.

Now, a description of the comments:

The `PostScriptFile` comment tells the driver to use the POST type resources contained in the file `FileNameString`. `FileNameString` is declared as a `Str255`.

When this comment is encountered, the driver calls `OpenResFile` using the file name specified in `FileNameString`. It then calls `GetResource('POST',theID);` repeatedly, where `theID` begins at 501 and is incremented by one for each `GetResource` call. If the driver gets a `ResNotFound` error, it closes the specified resource file. If the first byte of the resource is a 3, 4, or 5 then the remaining data is sent and the file is closed.

The format of the POST resource is as follows: The IDs of the resources start at 501 and are incremented by one for each resource. Each resource begins with a 2 byte data field containing the data type in the first byte and a zero in the second. The possible values for the first byte are:

0 ignore the rest of this resource (a comment)
1 data is ASCII text
2 data is binary and is first converted to ASCII before being sent
3 AppleTalk end of file. The rest of the data, if there is any, is interpreted as ASCII text and will be sent after the EOF.
4 open the data fork of the current resource file and send the ASCII text there
5 end of the resource file

The second byte of the field must always be zero. Resources should be kept small, around 2K. Text and binary should not be mixed in the same resource. Make sure you include either a space or a return at the end of each PostScript string to separate it from the following command.

Here's an example:

```
PROCEDURE PostFile;
{This procedure shows how to use PostScript from a specified FILE}
CONST
  PostScriptBegin = 190;
  PostScriptFile = 193;
  PostScriptEnd = 191;

VAR
  MyString          : Str255;
  MyHandle          : Handle;
  err               : OSErr;

BEGIN   { PostFile }
  {You should never do this in a real program. This is only a test.}
  MyString := 'HardDisk:MPW:Print Examples:PSTestDoc';
  err := PtrToHand(pointer(MyString),MyHandle,length(MyString) + 1);
  MyPic := OpenPicture(theWorld);
  ClipRect(theWorld);
  MoveTo(20,20);
  DrawString('PostScriptFile Comment');
  PicComment(PostScriptBegin,0,nil); {Begin PostScript}
  PicComment(PostScriptFile,GetHandleSize(MyHandle),MyHandle);
  PicComment(PostScriptEnd,0,nil); {PostScriptEnd}
  MoveTo(50,50);
  DrawString('PostScriptEnd has terminated');
  ClosePicture;
  DisposHandle(MyHandle); {Clean up}
  PrintthePicture; {print it please}
  KillPicture(MyPic);
END;    { PostFile }
```

Here are the resources:

```
type 'POST' {
  switch {
    case Comment:           /* this is a comment */
         key bitstring[8] = 0;
         fill byte;
         string;

    case ASCII:            /* this is just ASCII text */
         key bitstring[8] = 1;
         fill byte;
         string;

    case Bin:              /* this is binary */
         key bitstring[8] = 2;
         fill byte;
         string;

    case ATEOF:            /* this is an AppleTalk EOF */
         key bitstring[8] = 3;
         fill byte;
         string;
```

```
        case DataFork:          /* send the text in the data fork */
             key bitstring[8] = 4;
             fill byte;

        case EOF:               /* no more */
             key bitstring[8] = 5;
             fill byte;
        };
    };

    resource 'POST' (501) {
    ASCII{"0 728 translate "}};

    resource 'POST' (502) {
    ASCII{"1 -1 scale "}};

    resource 'POST' (503) {
    ASCII{"newpath "}};

    resource 'POST' (504) {
    ASCII{"100 470 moveto "}};

    resource 'POST' (505) {
    ASCII{"500 470 lineto "}};

    resource 'POST' (506) {
    ASCII{"100 330 moveto "}};

    resource 'POST' (507) {
    ASCII{"500 330 lineto "}};

    resource 'POST' (508) {
    ASCII{"230 600 moveto "}};

    resource 'POST' (509) {
    ASCII{"230 200 lineto "}};

    resource 'POST' (510) {
    ASCII{"370 600 moveto "}};

    resource 'POST' (511) {
    ASCII{"370 200 lineto "}};

    resource 'POST' (512) {
    ASCII{"10 setlinewidth "}};

    resource 'POST' (513) {
    ASCII{"stroke "}};

    resource 'POST' (514) {
    ASCII{"/Times-Roman findfont 12 scalefont setfont "}};

    resource 'POST' (515) {
    ASCII{"230 600 moveto "}};

    resource 'POST' (516) {
    ASCII{"(Hello World) show "}};
```

```
/* It will stop reading and close the file after 517 */
 resource 'POST' (517) {
 EOF
 {}};

/* it never gets here */
 resource 'POST' (518) {
 DataFork
 {}};
```

When the `ResourcePS` comment is encountered, the LaserWriter driver sends the text contained in the specified resource as PostScript to the printer. The additional data is defined as

```
PSRsrc = RECORD
           PSType : ResType;
           PSID   : INTEGER;
           PSIndex: INTEGER;
         END;
```

The resource can be of type STR or STR#. If the Type is STR then the index should be 0. Otherwise an index should be given.

This comment is essentially the same as the PrintF control call to the driver. The imbedded command string it uses is `'^r^n'`, which basically tells the driver to send the string specified by the additional data, then send a newline. For more information about printer control calls see the *LaserWriter Reference Manual*.

Here's an example:

```
PROCEDURE PostRSRC;
{This procedure shows how to get PostScript from a resource FILE}
  CONST
    PostScriptBegin = 190;
    PostScriptEnd = 191;
    ResourcePS = 195;

  TYPE
    theRSRChdl = ^theRSRCptr;
    theRSRCptr = ^theRSRC;
    theRSRC = RECORD
       theType: ResType;
       theID: INTEGER;
       Index: INTEGER;
    END;

  VAR
    temp         : Rect;
    TheResource  : theRSRChdl;
    i,j          : INTEGER;
    myport       : GrafPtr;
    err          : INTEGER;
    atemp        : Boolean;
```

```
BEGIN           { PostRSRC }
  TheResource := theRSRChdl(NewHandle(SizeOf(theRSRC)));
  TheResource^^.theID := 500;
  TheResource^^.Index := 0;
  TheResource^^.theType := 'STR ';
  HLock(Handle(TheResource));
  MyPic := OpenPicture(theWorld);
  DrawString('ResourcePS Comment');
  PicComment(PostScriptBegin,0,nil); {Begin PostScript}
  PicComment(ResourcePS,8,Handle(TheResource)); {Send postscript}
  PicComment(PostScriptEnd,0,nil); {PostScriptEnd}
  ClosePicture;
  DisposHandle(Handle(TheResource)); {Clean up}
  PrintthePicture; {print it please}
  KillPicture(MyPic);
END;            { PostRSRC }
```

Here's the resource:

```
resource 'STR ' (500)
{"0 728 translate 1 -1 scale newpath 100 470 moveto 500 470 lineto 100 330
moveto 500 330 lineto 230 600 moveto 230 200 lineto 370 600 moveto 370 200
lineto 10 setlinewidth stroke /Times-Roman findfont 12 scalefont setfont 230
600 moveto (Hello World) show"
};
```

Rotation

The concept of rotation doesn't apply to text alone. PostScript can rotate any object. The rotation comments work exactly like text rotation except that all objects drawn between the two comments are drawn in the rotated coordinate system specified by the center of rotation comment, not just text. Also, no clipping of `CopyBits` calls occurs. These comments only work on the 3.1 and newer LaserWriter drivers.

The `RotateBegin` comment tells the driver that the following objects will be drawn in a rotated plane. This comment contains the following data structure:

```
Rotation = RECORD
  Flip:  INTEGER; {0,1,2 => none, horizontal, vertical coordinate flip }
  Angle: INTEGER; {0..360 => clockwise rotation in degrees }
END; { Rotation }
```

When you are finished, the `RotateEnd` comment returns the coordinate system to normal, terminating the rotation.

The relative center of rotation is specified by the `RotateCenter` comment in exactly the same manner as the `TextCenter` comments. The difference, however, is that this comment **must** appear **before** the `RotateBegin` comment. The data structure of the accompanying handle is exactly like that for the `TextCenter` comment.

Here's an example of how to use rotation comments:

```
PROCEDURE Test;
{This procedure shows how to do rotations}
CONST
  RotateBegin = 200;
  RotateEnd = 201;
  RotateCenter = 202;

TYPE
    rothdl = ^rotptr;
    rotptr = ^trot;
    trot = RECORD
      flip : INTEGER;
      Angle : INTEGER;
    END; { trot }
    centhdl = ^centptr;
    centptr = ^cent;
    Cent = PACKED RECORD
             yInt: INTEGER;
             yFrac: INTEGER;
             xInt: INTEGER;
             xFrac: INTEGER;
    END; { Cent }

VAR
  arect    : Rect;
  rotation : rothdl;
  center   : centhdl;
```

```
BEGIN { Test }
  rotation := rothdl(NewHandle(sizeof(trot)));
  rotation^^.flip := 0;                              {no flip}
  rotation^^.angle := 15;       {15 degree rotation}

  center := centhdl(NewHandle(sizeof(cent)));
  center^^.xInt := 50;                               {center at 50,50}
  center^^.yInt := 50;
  center^^.xFrac := 0;                               {no fractional part}
  center^^.yFrac := 0;

  myPic := OpenPicture(theWorld);
    ClipRect(theWorld);
    MoveTo(20,20);
    DrawString('Begin Rotation');

    {set the center of Rotation}
    PicComment(RotateCenter,GetHandleSize(Handle(center)),Handle(center));
    {Begin Rotation}
   PicComment(RotateBegin,GetHandleSize(Handle(rotation)),Handle(rotation));
    SetRect(arect,100,100,500,500);
    FrameRect(aRect);
    MoveTo(500,500);
    Lineto(100,100);
    PicComment(RotateEnd,0,nil);             {RotateEnd}
  ClosePicture;
  DisposHandle(handle(rotation));            {Clean up}
  DisposHandle(handle(center));
  PrintThePicture;                                                    {print
it please}
  KillPicture(MyPic);
END; { Test }
```

Forms

The two form printing comments allow you to prepare a template to use for printing. When the `FormsBegin` comment is used, the LaserWriter's buffer is not cleared after `PrClosePage`. This allows you to download a form then change it for each subsequent page, inserting the information you want. `FormsEnd` allows the buffer to be cleared at the next `PrClosePage`.

#92: The Appearance of Text

See also: The Printing Manager
The Font Manager
Technical Note #91—
Optimizing for the LaserWriter—Picture Comments

Written by: Ginger Jernigan November 15, 1986
Updated: March 1, 1988

This technical note describes why text doesn't always look the way you expect depending on the environment you are in.

There are a number of Macintosh text editing applications where layout is critical. Unfortunately, text on a newer machine sometimes prints differently than text on a 64K ROM Macintosh. Let's examine some differences you should expect and why.

The differences we will consider here are only differences in the layout of text lines (line layout), not differences in the appearance of fonts or the differences between different printers. Differences in line layout may affect the position of line, paragraph and page breaks. The four variables that can affect line layout are fonts, the printer driver, the font manager mode, and ROMs.

Fonts

Every font on a Macintosh contains its own table of widths which tells QuickDraw how wide characters are on the screen. For every style point size there is a separate table which may contain widths that vary from face to face and from point size to point size. Character widths can vary between point sizes of characters even in the same face. In other words, fonts on the screen are not necessarily linearly scalable.

Non-linearity is not normally a problem since most fonts are designed to be as close to linear as possible. A font face in 6 point has very nearly the same scaled widths of the same font face in 24 point (plus or minus round-off or truncation differences). QuickDraw, however, requires only one face of any particular font to be in the System file to use it in any point size. If only a 10 point face actually exists, QuickDraw may scale that face to 9, 18, 24 (or whatever point size) by performing a linear scale of the 10 point face.

This can cause problems. Suppose a document is created on one Macintosh containing a font that only exists in that System file in one point size, say 9 point. The document is then taken to another Macintosh with a System file containing that same font but only in 24 point. The document may, in fact, appear differently on the two screens, and when it is printed, will have line breaks (and thus paragraph and page breaks) occurring in different places simply because of the differences in character widths that exist between the 9 point and 24 point faces.

The Printer Driver

Even when the printer you are using has a much higher resolution than what the screen can show, printer drivers perform line layout to match the screen layout as closely as possible.

The line layout performed by printer drivers is limited to single lines of text and does not change line break positions within multiple lines. The driver supplies metric information to the application about the page size and printable area to allow the application to determine the best place to make line and page breaks.

Printer driver line layout does affect word spacing, character spacing and even word positioning within a line. This may affect the overall appearance of text, particularly when font substitutions are made or various forms of page or text scaling are involved. But print drivers NEVER change line, paragraph or page break positions from what the application or screen specified. This means that where line breaks appear on the screen, they will always appear in the same place on the printer regardless of how the line layout may affect the appearance within the line.

Operating System and ROMs

In this context, operating system refers to the ROM trap routines which handle fonts and QuickDraw. Changes have occurred between the ROMs in the handling of fonts. Fonts in the 64K ROMs contain width tables (as described above) which are limited to integer values. Several new tables, however, have been added to fonts for the newer ROMs. The newer ROMs add an optional global width table containing fractional or fixed point decimal values. In addition, there is another optional table containing fractional values which can be scaled for the entire range of point sizes for any one face. There is also an optional table which provides for the addition (or removal) of width to a font when its style is changed to another value such as bold, outline or condensed. It is also possible, under the 128K ROMs, to add fonts to the system with inherent style properties containing their own width tables that produce different character widths from derived style widths.

One or all of the above tables may or may not be invoked depending on, first, their presence, and second, the mode of the operating system. The Font Manager in the newer ROMs allows the application to arbitrarily operate in either the fractional mode or integer mode (determined, in most cases, by the setting of `FractEnable`) as it chooses, with the default being integer. There is one case where fractional widths will be used if they exist even though fractional mode is disabled. When `FScaleDisable` is used fractional widths are always used if they exist regardless of the setting of `FractEnable`.

Differences in line layout (and thus line breaks) may be affected by any combination of the presence or absence of the optional tables, and the operating mode, either fractional or integer, of the application. Any of the combinations can produce different results from the original ROMs (and from each other).

The integer mode on the newer ROMs is very similar to, but not exactly the same as, the original 64K ROMs. When fonts with the optional tables present are used on Macintoshes with 64K ROMs, they continue to work in the old way with the integer widths. However, on newer ROMs, even in the integer mode, there may be variations in line width from what is seen on the old ROMs. In the plain text style there is very little if any difference (except if the global width table is present), but as various type styles are selected, line widths may vary more between ROMs.

Variations in the above options, by far, account for the greatest variation in the appearance of lines when a document is transported between one Macintosh and another. Line breaks may change position when documents created on one system (say a Macintosh) are moved to another system (like a Macintosh Plus). Variations are more pronounced as the number and sizes of various type styles increase within a document.

In all cases, however, a printer driver will produce exactly the same line breaks as appear on the screen with any given system combination.

#93: MPW: {$LOAD}; _DataInit;%_MethTables

See also: MPW Reference Manuals

Written by: Jim Friedlander November 15, 1986
Modified by: Jim Friedlander January 12, 1987
Updated: March 1, 1988

This technical note discusses the Pascal `{$LOAD}` directive as well as how to unload the `_DataInit` and `%_MethTables` segments.

{$LOAD}

MPW Pascal has a `{$LOAD}` directive that can dramatically speed up compiles.

```
{$LOAD HD:MPW:PLibraries:PasSymDump}
```

will combine symbol tables of all units following this directive (until another `{$LOAD}` directive is encountered), and dump them out to `HD:MPW:PLibraries:PasSymDump`. In order to avoid using fully specified pathnames, you can use `{$LOAD}` in conjunction with the `-k` option for Pascal:

```
Pascal -k "{PLibraries}" myfile
```

combined with the following lines in `myfile`:

```
USES
    {$LOAD PasSymDump}
        MemTypes,QuickDraw, OSIntf, ToolIntf, PackIntf,
    {$LOAD} {This "turns off" $LOAD for the next unit}
        NonOptimized,
    {$LOAD MyLibDump}
        MyLib;
```

will do the following: the first time a program containing these lines is compiled, two symbol table dump files (in this case `PasSymDump` and `MyLibDump`) will be created in the directory specified by the `-k` option (in this case `{PLibraries}`). No dump file will be generated for the unit `NonOptimized`. The compiler will compile `MemTypes`, `QuickDraw`, `OSIntf`, `ToolIntf`, `PackIntf` (quite time consuming) and dump those units' symbols to `PasSymDump` and it will compile the interface to `MyLib` and dump its symbols to `MyLib`. For subsequent compiles of this program (or any program that uses the same dump file(s)), the interface files won't be recompiled, the compiler will simply read in the symbol table.

Compiling a sample five line program on a Macintosh Plus/HD20SC takes 62 seconds without using the {$LOAD} directive. The same program takes 10 seconds to compile using the {$LOAD} directive (once the dump file exists). For further details about this topic, please see the *MPW Pascal Reference Manual* .

Note: If any of the units that are dumped into a dump file change, you need to make sure that the dump file is deleted, so that it can be regenerated by the Pascal compiler with the correct information. The best way to do this is to use a makefile to check the dump file against the files it depends on, and delete the dump file if it is out of date with respect to any of the units that it contains. An excellent (and well commented) example of doing this is in the *MPW Workshop Manual.*

The _DataInit Segment

The Linker will generate a segment whose resource name is %A5Init for any program compiled by the C or Pascal compilers. This segment is called by a program's main segment. This segment is loaded into the application heap and locked in place. It is up to your program to unload this segment (otherwise, it will remain locked in memory, possibly causing heap fragmentation). To do this from Pascal, use the following lines:

```
PROCEDURE _DataInit;EXTERNAL;
...

BEGIN         {main PROGRAM}
UnloadSeg(@_DataInit);
{remove data initialization code before any allocations}
...
```

From C, use the following lines:

```
extern _DataInit();
...
{ /* main */
              UnloadSeg(_DataInit);
              /*remove data initialization code before any allocations*/
...
```

For further details about Data Initialization, see the *MPW Reference Manual.*

%_MethTables and %_SelProcs

Object use in Pascal produces two segments which can cause heap problems. These are `%_MethTables` and `%_SelProcs` which are used when method calls are made. MacApp deals with them correctly, so this only applies to Object Pascal programs that don't use MacApp. You can make the segments locked and preloaded (probably the easiest route), so they will be loaded low in the heap, or you can unload them temporarily while you are doing heap initialization. In the latter case, make sure there are no method calls while they are unloaded. To reload `%_MethTables` and `%_SelProcs`, call the dummy procedure `%_InitObj`. `%_InitObj` loads `%MethTables` —calling any method will then load `%_SelProcs`.

Reminder: The linker is case sensitive when dealing with module names. Pascal converts all module names to upper-case (unless a routine is declared to be a C routine). The Assembler default is the same as the Pascal default, though it can be changed with the CASE directive. C preserves the case of module names (unless a routine is declared to be `pascal`, in which case the module name is converted to upper-case letters).

Make sure that any external routines that you reference are capitalized the same in both the external routine and the external declaration (especially in C). If the capitalization differs, you will get the following link error (library routine = `findme`, program declaration = `extern FindMe();`):

```
### Link: Error  Undefined entry, name: FindMe
```

#94: Tags

See also: The File Manager

Written by: Bryan Stearns
Updated:

November 15, 1986
March 1, 1988

Apple has decided to eliminate support for file-system tags on its future products; this technical note explains this decision.

Some of Apple's disk products (and some third-party products) have the ability to store 532 bytes per sector, instead of the normal 512. Twelve of the extra bytes are used to store redundant file system information, known as "tags", to be used by a scavenging utility to reconstruct damaged disks.

Apple has decided to eliminate support for these tags on its products; this was decided for several reasons:

1) Tags were implemented back when we had to deal with "Twiggy" drives on Lisa. These drives were less reliable than current drives, and it was expected that tags would be needed for data integrity.

2) We're working on a scavenging utility (Disk First Aid), and we've found that tags don't help us in reconstructing damaged disks (ie, if we can't fix it without using tags, tags wouldn't help us fix it). So, at least the first two versions of our scavenging utility will not use tags, and a third version (which we've planned for, but will probably never implement) can probably work without them.

3) 532-byte-per-sector drives and controllers tend to cost more, even at Apple's volumes. Thus, the demise of tags saves us (and our customers) money. The Apple Hard Disk 20SC currently supports tags; this may not always be the case, however; we'll probably drop the large sectors when we run out of our current stock of drives.

The Hierarchical File System (HFS) documentation didn't talk about tags because the writer had no information available about how they worked under HFS. Because of this decision, it is unlikely that we'll ever have documentation on how to correctly implement them under HFS.

#95: How To Add Items to the Print Dialogs

See also: The Printing Manager
The Dialog Manager

Written by: Ginger Jernigan
Lew Rollins

November 15, 1986

Updated: March 1, 1988

This technical note discusses how to add your own items to the Printing Manager's dialogs.

When the Printing Manager was initially designed, great care was taken to make the interface to the printer drivers as generic as possible in order to allow applications to print without being device-specific. There are times, however, when this type of non-specific interface interferes with the flexibility of an application. An application may require additional information before printing which is not part of the general Printing Manager interface. This technical note describes a method that an application can use to add its own items to the existing style and job dialogs.

Before continuing, you need to be aware of some guidelines that will increase your chances of being compatible with the printing architecture in the future:

- Only add items to the dialogs as described in this technical note. Any other methods will decrease your chances of survival in the future.

- Do not change the position of any item in the current dialogs. This means don't delete items from the existing item list or add items in the middle. Add items **only at the end** of the list.

- Don't count on an item retaining its current position in the list. If you depend on the Draft button being a particular number in the ImageWriter's style dialog item list, and we change the Draft button's item number for some reason, your program may no longer function correctly.

- Don't use more than half the screen height for your items. Apple reserves the right to expand the items in the standard print dialogs to fill the top half of the screen.

- If you are adding lots of items to the dialogs (which may confuse users), you should consider having your own separate dialog in addition to the existing Printing Manager dialogs.

The Heart

Before we talk about how the dialogs work, you need to know this: at the heart of the printer dialogs is a little-known data structure partially documented in the MacPrint interface file. It's a record called `TPrDlg` and it looks like this:

```
TPrDlg = RECORD   {Print Dialog: The Dialog Stream object.}
   dlg       : DialogRecord;   {dialog window}
   pFltrProc : ProcPtr;        {filter proc.}
   pItemProc : ProcPtr;        {item evaluating proc.}
   hPrintUsr : THPrint;        {user's print record.}
   fDoIt     : BOOLEAN;
   fDone     : BOOLEAN;
   lUser1    : LONGINT;        {four longs reserved by Apple}
   lUser2    : LONGINT;
   lUser3    : LONGINT;
   lUser4    : LONGINT;
   iNumFst   : INTEGER;        {numeric edit items for std filter}
   iNumLst   : INTEGER;
{... plus more stuff needed by the particular printing dialog.}
END;
TPPrDlg = ^TPrDlg;             {== a dialog ptr}
```

All of the information pertaining to a print dialog is kept in the `TPrDlg` record. This record will be referred to frequently in the discussion below.

How the Dialogs Work

When your application calls `PrStlDialog` and `PrJobDialog`, the printer driver actually calls a routine called `PrDlgMain`. This function is declared as follows:

```
FUNCTION PrDlgMain (hprint: THPrint; pDlgInit: ProcPtr): BOOLEAN;
```

`PrDlgMain` first calls the `pDlgInit` routine to set up the appropriate dialog (in `Dlg`), dialog hook (`pItemProc`) and dialog event filter (`pFilterProc`) in the `TPrDlg` record (shown above). For the job dialog, the address of `PrJobInit` is passed to `PrDlgMain`. For the style dialog, the address of `PrStlInit` is passed. These routines are declared as follows:

```
FUNCTION PrJobInit (hPrint: THPrint): TPPrDlg;
FUNCTION PrStlInit (hPrint: THPrint): TPPrDlg;
```

After the initialization routine sets up the `TPrDlg` record, `PrDlgMain` calls `ShowWindow` (the window is initially invisible), then it calls `ModalDialog`, using the dialog event filter pointed to by the `pFltrProc` field. When an item is hit, the routine pointed to by the `pItemProc` field is called and the items are handled appropriately. When the OK button is hit (this includes pressing Return or Enter) the print record is validated. The print record is not validated if the Cancel button is hit.

How to Add Your Own Items

To modify the print dialogs, you need to change the `TPrDlg` record before the dialog is drawn on the screen. You can add your own items to the item list, replace the addresses of the standard dialog hook and event filter with the addresses of your own routines and then let the dialog code continue on its merry way.

For example, to modify the job dialog, first call `PrJobInit`. `PrJobInit` will fill in the `TPrDlg` record for you and return a pointer to that record. Then call `PrDlgMain` directly, passing in the address of your own initialization function. The example code's initialization function adds items to the dialog item list, saves the address of the standard dialog hook (in our global variable `prPItemProc`) and puts the address of our dialog hook into the `pItemProc` field of the `TPrDlg` record. Please note that your dialog hook **must** call the standard dialog hook to handle all of the standard dialog's items.

Note: If you wish to have an event filter, handle it the same way that you do a dialog hook.

Now, here is an example (written in MPW Pascal) that modifies the job dialog. The same code works for the style dialog if you globally replace 'Job' with 'Stl'. Also included is a function (`AppendDITL`) provided by Lew Rollins (originally written in C, translated for this technical note to MPW Pascal) which demonstrates a method of adding items to the item list, placing them in an appropriate place, and expanding the dialog window's rectangle.

The MPW Pascal Example Program

```
PROGRAM ModifyDialogs;

 USES
  {$LOAD PasDump.dump}
  MemTypes,QuickDraw,OSIntf,ToolIntf,PackIntf,MacPrint;

 CONST
  MyDITL    = 256;
  MyDFirstBox  = 1;     {Item number of first box in my DITL}
  MyDSecondBox = 2;

 VAR
  PrtJobDialog: TPPrDlg;   { pointer to job dialog }
  hPrintRec : THPrint;   { Handle to print record }
  FirstBoxValue,       { value of our first additional box }
  SecondBoxValue: Integer; { value of our second addtl. box }
  prFirstItem,       { save our first item here }
  prPItemProc : LongInt;   { we need to store the old itemProc here }
  itemType   : Integer;   { needed for GetDItem/SetDItem calls }
  itemH    : Handle;
  itemBox   : Rect;
  err     : OSErr;
  {-------------------------------------------------------------------------}

 PROCEDURE _DataInit;
  EXTERNAL;
```

```
 {------------------------------------------------------------------------}

 PROCEDURE CallItemHandler(theDialog: DialogPtr; theItem: Integer; theProc:
LongInt);
   INLINE $205F,$4E90;    { MOVE.L (A7)+,A0
                            JSR (A0)         }

{ this code pops off theProc and then does a JSR to it, which puts the
 real return address on the stack. }

 {------------------------------------------------------------------------}

 FUNCTION AppendDITL(theDialog: DialogPtr; theDITLID: Integer): Integer;
 { version 0.1 9/11/86 Lew Rollins of Human-Systems Interface Group}
 { this routine still needs some error checking }

{ This routine appends all of the items of a specified DITL
onto the end of a specified DLOG - We don't even need to know the format
of the DLOG }

{ this will be done in 3 steps:
 1. append the items of the specified DITL onto the existing DLOG
 2. expand the original dialog window as required
 3. return the adjusted number of the first new user item
}
   TYPE
    DITLItem    = RECORD { First, a single item }
            itmHndl: Handle; { Handle or procedure pointer for this item }
            itmRect: Rect; { Display rectangle for this item }
            itmType: SignedByte; { Item type for this item - 1 byte }
            itmData: ARRAY [0..0] OF SignedByte; { Length byte of data }
          END;   {DITLItem}

    pDITLItem   = ^DITLItem;
    hDITLItem   = ^pDITLItem;

    ItemList    = RECORD { Then, the list of items }
            dlgMaxIndex: Integer; { Number of items minus 1 }
            DITLItems: ARRAY [0..0] OF DITLItem; { Array of items }
          END;   {ItemList}

    pItemList   = ^ItemList;
    hItemList   = ^pItemList;

    IntPtr    = ^Integer;

   VAR
    offset   : Point;  { Used to offset rectangles of items being appended }
    maxRect   : Rect;   { Used to track increases in window size }
    hDITL    : hItemList; { Handle to DITL being appended }
    pItem    : pDITLItem; { Pointer to current item being appended }
    hItems   : hItemList; { Handle to DLOG's item list }
    firstItem : Integer; { Number of where first item is to be appended }
    newItems,       { Count of new items }
    dataSize,       { Size of data for current item }
    i      : Integer; { Working index }
    USB     : RECORD  {we need this because itmData[0] is unsigned}
            CASE Integer OF
```

```
            1:
              (SBArray: ARRAY [0..1] OF SignedByte);
            2:
              (Int: Integer);
          END;   {USB}

  BEGIN          {AppendDITL}
{
 Using the original DLOG

 1. Remember the original window Size.
 2. Set the offset Point to be the bottom of the original window.
 3. Subtract 5 pixels from bottom and right, to be added
    back later after we have possibly expanded window.
 4. Get working Handle to original item list.
 5. Calculate our first item number to be returned to caller.
 6. Get locked Handle to DITL to be appended.
 7. Calculate count of new items.
}
    maxRect := DialogPeek(theDialog)^.window.port.portRect;
    offset.v := maxRect.bottom;
    offset.h := 0;
    maxRect.bottom := maxRect.bottom - 5;
    maxRect.right := maxRect.right - 5;
    hItems := hItemList(DialogPeek(theDialog)^.items);
    firstItem := hItems^^.dlgMaxIndex + 2;
    hDITL := hItemList(GetResource('DITL',theDITLID));
    HLock(Handle(hDITL));
    newItems := hDITL^^.dlgMaxIndex + 1;
{
 For each item,
  1. Offset the rectangle to follow the original window.
  2. Make the original window larger if necessary.
  3. fill in item Handle according to type.
}

    pItem := @hDITL^^.DITLItems;
    FOR i := 1 TO newItems DO BEGIN
     OffsetRect(pItem^.itmRect,offset.h,offset.v);
     UnionRect(pItem^.itmRect,maxRect,maxRect);

     USB.Int := 0;    {zero things out}
     USB.SBArray[1] := pItem^.itmData[0];

     { Strip enable bit since it doesn't matter here. }
     WITH pItem^ DO
      CASE BAND(itmType,$7F) OF
       userItem:   { Can't do anything meaningful with user items. }
        itmHndl := NIL;
       ctrlItem + btnCtrl,ctrlItem + chkCtrl,ctrlItem + radCtrl:{build Control }
        itmHndl := Handle(NewControl(theDialog, { theWindow }
                     itmRect, { boundsRect }
                     StringPtr(@itmData[0])^, { title }
                     true, { visible }
                     0,0,1, { value, min, max }
                     BAND(itmType,$03), { procID }
                     0)); { refCon }
       ctrlItem + resCtrl: BEGIN { Get resource based Control }
```

```
        itmHndl := Handle(GetNewControl(IntPtr(@itmData[1])^, { controlID }
                          theDialog)); { theWindow }
        ControlHandle(itmHndl)^^.contrlRect := itmRect; {give it the right
                                rectangle}
        {An actionProc for a Control should be installed here}
      END;      {Case ctrlItem + resCtrl}
      statText,editText: { Both need Handle to a copy of their text. }
        err := PtrToHand(@itmData[1], { Start of data }
                 itmHndl, { Address of new Handle }
                 USB.Int); { Length of text }
      iconItem:   { Icon needs resource Handle. }
        pItem^.itmHndl := GetIcon(IntPtr(@itmData[1])^); { ICON resID }
      picItem:    { Picture needs resource Handle. }
        pItem^.itmHndl := Handle(GetPicture(IntPtr(@itmData[1])^));{PICT resID}
      OTHERWISE
        itmHndl := NIL;
    END;        {Case}

    dataSize := BAND(USB.Int + 1,$FFFE);
    {now advance to next item}
    pItem := pDITLItem(Ptr(ord4(@pItem^) + dataSize + sizeof(DITLItem)));
  END;          {for}
  err := PtrAndHand
    (@hDITL^^.DITLItems,Handle(hItems),GetHandleSize(Handle(hDITL)));
  hItems^^.dlgMaxIndex := hItems^^.dlgMaxIndex + newItems;
  HUnlock(Handle(hDITL));
  ReleaseResource(Handle(hDITL));
  maxRect.bottom := maxRect.bottom + 5;
  maxRect.right := maxRect.right + 5;
  SizeWindow(theDialog,maxRect.right,maxRect.bottom,true);
  AppendDITL := firstItem;
END;            {AppendDITL}

{-------------------------------------------------------------------------}

PROCEDURE MyJobItems(theDialog: DialogPtr; itemNo: Integer);
{
This routine replaces the routine in the pItemProc field in the
TPPrDlg record.  The steps it takes are:
1. Check to see if the item hit was one of ours. This is done by "localizing"
   the number, assuming that our items are numbered from 0..n
2. If it's one of ours  then case it and Handle appropriately
3. If it isn't one of ours then call the old item handler
}

  VAR
    MyItem,firstItem: Integer;
    thePt    : Point;
    thePart   : Integer;
    theValue   : Integer;
    debugPart  : Integer;

  BEGIN          {MyJobItems}
    firstItem := prFirstItem; { remember, we saved this in myJobDlgInit }
    MyItem := itemNo - firstItem + 1; { "localize" current item No }
    IF MyItem > 0 THEN BEGIN { if localized item > 0, it's one of ours }
      { find out which of our items was hit }
      GetDItem(theDialog,itemNo,itemType,itemH,itemBox);
```

```
      CASE MyItem OF
       MyDFirstBox: BEGIN
         { invert value of FirstBoxValue and redraw it }
         FirstBoxValue := 1 - FirstBoxValue;
         SetCtlValue(ControlHandle(itemH),FirstBoxValue);
       END;       {case MyDFirstBox}
       MyDSecondBox: BEGIN
         { invert value of SecondBoxValue and redraw it }
         SecondBoxValue := 1 - SecondBoxValue;
         SetCtlValue(ControlHandle(itemH),SecondBoxValue);
       END;       {case MyDSecondBox}
       OTHERWISE
         Debug;     { OH OH - We got an item we didn't expect }
      END;         {Case}
    END           { if MyItem > 0 }
    ELSE         { chain to standard item handler, whose address is saved
                  in prPItemProc }
      CallItemHandler(theDialog,itemNo,prPItemProc);
  END;            { MyJobItems }

 {-------------------------------------------------------------------------}

 FUNCTION MyJobDlgInit(hPrint: THPrint): TPPrDlg;
{
 This routine appends items to the standard job dialog and sets up the
 user fields of the printing dialog record TPRDlg
 This routine will be called by PrDlgMain
 This is what it does:
 1. First call PrJobInit to fill in the TPPrDlg record.
 2. Append our items onto the old DITL. Set them up appropriately.
 3. Save the address of the old item handler and replace it with ours.
 4. Return the Fixed dialog to PrDlgMain.
 }

  VAR
    firstItem : Integer; { first new item number }

  BEGIN           {MyJobDlgInit}
    firstItem := AppendDITL(DialogPtr(PrtJobDialog),MyDITL);

    prFirstItem := firstItem; { save this so MyJobItems can find it }

    { now we'll set up our DITL items - The "First Box" }
    GetDItem(DialogPtr(PrtJobDialog),firstItem,itemType,itemH,itemBox);
    SetCtlValue(ControlHandle(itemH),FirstBoxValue);

    { now we'll set up the second of our DITL items - The "Second Box" }
    GetDItem(DialogPtr(PrtJobDialog),firstItem + 1,itemType,itemH,itemBox);
    SetCtlValue(ControlHandle(itemH),SecondBoxValue);

{ Now comes the part where we patch in our item handler.  We have to save
 the old item handler address, so we can call it if one of the standard
 items is hit, and put our item handler's address
 in pItemProc field of the TPrDlg struct}

    prPItemProc := LongInt(PrtJobDialog^.pItemProc);

    { Now we'll tell the modal item handler where our routine is }
```

```
    PrtJobDialog^.pItemProc := ProcPtr(@MyJobItems);

    { PrDlgMain expects a pointer to the modified dialog to be returned.... }
    MyJobDlgInit := PrtJobDialog;

  END;            {myJobDlgInit}

 {-------------------------------------------------------------------------}

 FUNCTION Print: OSErr;

  VAR
    bool     : BOOLEAN;

  BEGIN           {Print}
    hPrintRec := THPrint(NewHandle(sizeof(TPrint)));
    PrintDefault(hPrintRec);
    bool := PrValidate(hPrintRec);
    IF (PrError <> noErr) THEN BEGIN
      Print := PrError;
      Exit(Print);
    END;          {If}

    { call PrJobInit to get pointer to the invisible job dialog }
    PrtJobDialog := PrJobInit(hPrintRec);
    IF (PrError <> noErr) THEN BEGIN
      Print := PrError;
      Exit(Print);
    END;          {If}

 {Here's the line that does it all!}
    IF NOT (PrDlgMain(hPrintRec,@MyJobDlgInit)) THEN BEGIN
      Print := cancel;
      Exit(Print);
    END;          {If}

    IF PrError <> noErr THEN Print := PrError;

    { that's all for now }

  END;            { Print }

 {-------------------------------------------------------------------------}

 BEGIN            {PROGRAM}

   UnloadSeg(@_DataInit);   {remove data initialization code before any
allocations}
    InitGraf(@thePort);
    InitFonts;
    FlushEvents(everyEvent,0);
    InitWindows;
    InitMenus;
    TEInit;
    InitDialogs(NIL);
    InitCursor;

    { call the routine that does printing }
```

```
  FirstBoxValue := 0;     { value of our first additional box }
  SecondBoxValue := 0;    { value of our second addtl. box }
  PrOpen;     { Open the Print Manager }
  IF PrError = noErr THEN
    err := Print      { This actually brings up the modified Job dialog }
  ELSE BEGIN
    {tell the user that PrOpen failed}
  END;

  PrClose;     { Close the Print Manager and leave }
END.
```

The Lightspeed C Example Program

```
/* NOTE: Apple reserves the top half of the screen (where the current DITL
      items are located). Applications may use the bottom half of the
      screen to add items, but should not change any items in the top half
      of the screen.  An application should expand the print dialogs only
      as much as is absolutely necessary.
*/

/* Note: A global search and replace of 'Job' with 'Stl' will produce
      code that modifies the style dialogs */
#include <DialogMgr.h>
#include <MacTypes.h>
#include <Quickdraw.h>
#include <ResourceMgr.h>
#include <WindowMgr.h>
#include <pascal.h>
#include <printmgr.h>
#define nil 0L


static TPPrDlg PrtJobDialog;        /* pointer to job dialog */

/*    This points to the following structure:

        struct {
              DialogRecord    Dlg;        (The Dialog window)
              ProcPtr         pFltrProc;  (The Filter Proc.)
              ProcPtr         pItemProc;  (The Item evaluating proc. --
                                          we'll change this)
              THPrint         hPrintUsr;  (The user's print record.)
              Boolean         fDoIt;
              Boolean         fDone;
                  (Four longs -- reserved by Apple Computer)
              long                    lUser1;
              long                    lUser2;
              long                    lUser3;
              long                    lUser4;
   } TPrDlg; *TPPrDlg;
*/


/*    Declare 'pascal' functions and procedures */
pascal Boolean PrDlgMain();         /* Print manager's dialog handler */
pascal TPPrDlg PrJobInit();         /* Gets standard print job dialog. */
pascal TPPrDlg MyJobDlgInit();      /* Our extention to PrJobInit */
pascal void MyJobItems();           /* Our modal item handler */

#define MyDITL 256                  /* resource ID of my DITL to be spliced
                                          on to job dialog */


THPrint hPrintRec;                  /* handle to print record */
short FirstBoxValue = 0;            /* value of our first additional box */
short SecondBoxValue = 0;           /* value of our second addtl. box */
long prFirstItem;                   /* save our first item here */
long prPItemProc;             /* we need to store the old itemProc here */
```

```
/*-------------------------------------------------------------------------*/
      WindowPtr   MyWindow;
      OSErr       err;
      Str255      myStr;
main()
{
      Rect        myWRect;

      InitGraf(&thePort);
      InitFonts();
      InitWindows();
      InitMenus();
      InitDialogs(nil);
      InitCursor();
      SetRect(&myWRect,50,260,350,340);

      /* call the routine that does printing */
      PrOpen();
      err = Print();

      PrClose();
} /* main */

/*-------------------------------------------------------------------------*
/

OSErr Print()

{
      /* call PrJobInit to get pointer to the invisible job dialog */
      hPrintRec = (THPrint)(NewHandle(sizeof(TPrint)));
      PrintDefault(hPrintRec);
      PrValidate(hPrintRec);
      if (PrError() != noErr)
            return PrError();

      PrtJobDialog = PrJobInit(hPrintRec);
      if (PrError() != noErr)
            return PrError();


      if (!PrDlgMain(hPrintRec, &MyJobDlgInit)) /* this line does all the
                                                   stuff */
            return Cancel;

      if (PrError() != noErr)
            return PrError();

/* that's all for now */

} /* Print */

/*-------------------------------------------------------------------------*
/

pascal TPPrDlg MyJobDlgInit (hPrint)
THPrint hPrint;
```

```
/* this routine appends items to the standard job dialog and sets up the
      user fields of the printing dialog record TPRDlg
      This routine will be called by PrDlgMain */
{
      short       firstItem;        /* first new item number */

      short       itemType;         /* needed for GetDItem/SetDItem call */
      Handle      itemH;
      Rect        itemBox;

      firstItem = AppendDITL (PrtJobDialog, MyDITL); /*call routine to do
                                                       this */

      prFirstItem = firstItem; /* save this so MyJobItems can find it */

/* now we'll set up our DITL items -- The "First Box" */
      GetDItem(PrtJobDialog,firstItem,&itemType,&itemH,&itemBox);
      SetCtlValue(itemH,FirstBoxValue);

/* now we'll set up the second of our DITL items  -- The "Second Box" */
      GetDItem(PrtJobDialog,firstItem+1,&itemType,&itemH,&itemBox);
      SetCtlValue(itemH,SecondBoxValue);

/* Now comes the part where we patch in our item handler.  We have to save
      the old item handler address, so we can call it if one of the
      standard items is hit, and put our item handler's address
      in pItemProc field of the TPrDlg struct
*/

      prPItemProc = (long)PrtJobDialog->pItemProc;

/* Now we'll tell the modal item handler where our routine is */
      PrtJobDialog->pItemProc = (ProcPtr)&MyJobItems;

/* PrDlgMain expects a pointer to the modified dialog to be returned.... */
      return PrtJobDialog;

} /*myJobDlgInit*/


/*--------------------------------------------------------------------------*/

/* here's the analogue to the SF dialog hook */

pascal void MyJobItems(theDialog,itemNo)
TPPrDlg     theDialog;
short       itemNo;

{ /* MyJobItems */
      short       myItem;
      short       firstItem;

      short       itemType;         /* needed for GetDItem/SetDItem call */
      Handle      itemH;
      Rect        itemBox;

      firstItem = prFirstItem; /* remember, we saved this in myJobDlgInit
*/
```

```
        myItem = itemNo-firstItem+1;  /* "localize" current item No */
        if (myItem > 0)    /* if localized item > 0, it's one of ours */
        {
              /* find out which of our items was hit */
              GetDItem(theDialog,itemNo,&itemType,&itemH,&itemBox);
              switch (myItem)
              {
                    case 1:
                          /* invert value of FirstBoxValue and redraw it */
                          FirstBoxValue ^= 1;
                          SetCtlValue(itemH,FirstBoxValue);
                          break;

                    case 2:
                          /* invert value of SecondBoxValue and redraw it */
                          SecondBoxValue ^= 1;
                          SetCtlValue(itemH,SecondBoxValue);
                          break;
                    default: Debugger(); /* OH OH */
              } /* switch */
        } /* if (myItem > 0) */
        else /* chain to standard item handler, whose address is saved in
                    prPItemProc */
        {
              CallPascal(theDialog,itemNo,prPItemProc);
        }
  } /* MyJobItems */
```

The Rez Source

```
#include "types.r"

resource 'DITL' (256) {
  {  /* array DITLarray: 2 elements */
    /* [1] */
    {8, 0, 24, 112},
    CheckBox {
      enabled,
      "First Box"
    };
    /* [2] */
    {8, 175, 24, 287},
    CheckBox {
      enabled,
      "Second Box"
    }
  }
};
```

#96: SCSI Bugs

See also: The SCSI Manager
SCSI Developer's Package

Written by: Steve Flowers — October 1, 1986
Modified by: Bryan Stearns — November 15, 1986
Modified by: Bo3b Johnson — July 1, 1987
Updated: — March 1, 1988

There are a number of problems in the SCSI Manager; this note lists the ones we know about, along with an explanation of what we're doing about them. Changes made for the 2/88 release are made to more accurately reflect the state of the SCSI Manager. System 4.1 and 4.2 are very similar; one bug was fixed in System 4.2.

There are several categories of SCSI Manager problems:

1. Those in the ROM boot code
(Before the System file has been opened, and hence, before any patches could possibly fix them.)
2. Those that have been fixed in System 3.2
3. Those that have been fixed in System 4.1/4.2
4. Those that are new in System 4.1/4.2
5. Those that have not yet been fixed.

The problems in the ROM boot code can only be fixed by changing the ROMs. Most of the bugs in the SCSI Manager itself have been fixed by the patch code in the System 3.2 file. There are a few problems, though, that are not fixed with System 3.2—most of these bugs have been corrected in System 4.1/4.2. Any that are not fixed will be detailed here. ROM code for future machines will, of course, include the corrections.

ROM boot code problems

- In the process of looking for a bootable SCSI device, the boot code issues a SCSI bus reset before each attempt to read block 0 from a device. If the read fails for any reason, the boot code goes on to the next device. SCSI devices which implement the `Unit Attention` condition as defined by the Revision 17B SCSI standard will fail to boot in this case. The read will fail because the drive is attempting to report the `Unit Attention` condition for the first command it receives after the SCSI bus reset. The boot code does not read the sense bytes and does not retry the failed command; it simply resets the SCSI bus and goes on to the next device.

If no other device is bootable, the boot code will eventually cycle back to the same SCSI device ID, reset the bus (causing `Unit Attention` in the drive again), and try to read block 0 (which fails for the same reason).

The 'new' Macintosh Plus ROMs that are included in the platinum Macintosh Plus have only one change. The change was to simply do a single SCSI Bus Reset after power up instead of a Reset each time through the SCSI boot loop. This was done to allow `Unit Attention` drives to be bootable. It was an object code patch (affecting approximately 30 bytes) and no other bugs were fixed. For details on the three versions of Macintosh Plus ROMs, see Technical Note #154.

We recommend that you choose an SCSI controller which does not require the `Unit Attention` feature—either an older controller (most of the SCSI controllers currently available were designed before Revision 17B), or one of the newer Revision-17B-compatible controllers which can enable/disable `Unit Attention` as a formatting option (such as those from Seagate, Rodime, et al). Since the vast majority of Macintosh Plus computers have the ROMs which cannot use `Unit Attention` drives, we still recommend that you choose an SCSI controller that does not require the `Unit Attention` feature.

- If an SCSI device goes into the `Status` phase after being selected by the boot code, this leads to the SCSI bus being left in the `Status` phase indefinitely, and no SCSI devices can be accessed. The current Macintosh Plus boot code does not handle this change to `Status` phase, which means that the presence of an SCSI device with this behavior (as in some tape controllers we've seen) will prevent any SCSI devices from being accessed by the SCSI Manager, even if they already had drivers loaded from them. The result is that any SCSI peripheral that is turned on at boot time must not go into `Status` phase immediately after selection; otherwise, the Macintosh Plus SCSI bus will be left hanging. Unless substantially revised ROMs are released for the Macintosh Plus (highly unlikely within the next year or so), this problem will never be fixed on the Macintosh Plus, so you should design for old ROMs.

- The Macintosh Plus would try to read 256 bytes of blocks 0 and 1, ignoring the extra data. The Macintosh SE and Macintosh II try to read 512 bytes from blocks 0 and 1, ignoring errors if the sector size is larger (but not smaller) than 512 bytes. Random access devices (disks, tapes, CD ROMS, etc.) can be booted as long as the blocks are at least 512 bytes, blocks 0, 1 and other partition blocks are correctly set up, and there is a driver on it. With the new partition layout (documented in *Inside Macintosh* volume V), more than 256 bytes per sector may be required in some partition map entries. This is why we dropped support for 256-byte sectors. Disks with tag bytes (532-byte sectors) or larger block sizes (1K, 2K, etc.) can be booted on any Macintosh with an SCSI port. Of course, the driver has to take care of data blocking and de-blocking, since HFS likes to work with 512-byte sectors.

Problems with ROM SCSI Manager routines

Note that the following problems are fixed after the System file has been opened; for a device to boot properly, it must not depend on these fixes. The sample SCSI driver, available from APDA, contains an example of how to find out if the fixes are in place.

- **Prior to System file 3.2,** blind transfers (both reads and writes) would not work properly with many SCSI controllers. Since blind operation depends on the drive's ability to transfer data fast enough, it is the responsibility of the driver writer to make sure blind operation is safe for a particular device.

- **Prior to System file 3.2,** the SCSI Manager dropped a byte when the driver did two or more `SCSIReads` or `SCSIRBlinds` in a row. (Each `Read` or `RBlind` has to have a Transfer Information Block (TIB) pointer passed in.) The TIB itself can be as big and complex as you want—it is the process of returning from one `SCSIRead` or `SCSIRBlind` and entering another one (while still on the same SCSI command) that causes the first byte for the other `SCSIReads` to be lost.

  Note that this precludes use of file-system tags. Apple no longer recommends that you support tags; see Technical Note #94 for more information.

- **Prior to System file 3.2,** `SCSIStat` didn't work; the new version works correctly.

- **Running under System file 3.2,** the SCSI Manager does not check to make sure that the last byte of a write operation (to the peripheral) was handshaked while operating in pseudo-DMA mode. The SCSI Manager writes the final byte to the NCR 5380's one-byte buffer and then turns pseudo-DMA mode off shortly thereafter (reported to be 10-15 microseconds). If the peripheral is somewhat slow in actually reading the last byte of data, it asserts `REQ` after the Macintosh has already turned off pseudo-DMA mode and never gets an `ACK`. The CPU then expects to go into the `Status` phase since it thinks everything went OK, but the peripheral is still waiting for `ACK`. Unless the driver can recover from this somehow, the SCSI bus is 'hung' in the `Data Out` phase. In this case, all successive SCSI Manager calls will fail until the bus is reset.

- **Running under System file 4.1/4.2,** the SCSI Manager waits for the last byte of a write operation to be handshaked while operating in pseudo-DMA mode; it checks for a final `DRQ` (or a phase change) at the end of a `SCSIWrite` or `SCSIWBlind` before turning off the pseudo-DMA mode. Drivers that could recover from this problem by writing the last byte again if the bus was still in a `Data Out` phase will still work correctly, as long as they were checking the bus state.

- **Running under System file 3.2,** the SCSI Manager does not time out if the peripheral fails to finish transferring the expected number of bytes for polled reads and writes. (Blind operation does poll for the first byte of each requested data transfer in the Transfer Information Block.)

- **Running under System file 4.1/4.2,** `SCSIRead` and `SCSIWrite` return an error to the caller if the peripheral changes the bus phase in the middle of a transfer, as might happen if the peripheral fails to transfer the expected number of bytes. The computer is no longer left in a hung state.

- **Running under System file 3.2,** the Selection timeout value is very short (900 microseconds). Patches to the SCSI Manager **in System 4.1/4.2** ensure that this value is the recommended 250 milliseconds.

- **Running under System file 3.2,** the SCSI Manager routine `SCSIGet` (which arbitrates for the bus) will fail if the `BSY` line is still asserted. Some devices are a bit slow in releasing `BSY` after the completion of an SCSI operation, meaning that `BSY` may not have been released before the driver issues a `SCSIGet` call to start the next SCSI operation. A work-around for this is to call `SCSIGet` again if it failed the first time. (Rarely has it been necessary to try it a third time.) This assumes, of course, that the bus has not been left 'hanging' by an improperly terminated SCSI operation before calling `SCSIGet`.

- **Running under System file 4.1/4.2,** the `SCSIGet` function has been made more tolerant of devices that are slow to release the `BSY` line after a SCSI operation. The SCSI Manager now waits up to 200 milliseconds before returning an error.

Problems with the SCSI Manager that haven't been fixed yet

These problems currently exist in the Macintosh Plus, SE, and II SCSI Manager. We plan to fix these problems in a future release of the System Tools disk, but in the mean time, you should try to work around the problems (but don't "require" the problems!).

- Multiple calls to `SCSIRead` or `SCSIRBlind` after issuing a command and before calling `SCSIComplete` may not work. Suppose you want to read some mode sense data from the drive. After sending the command with `SCSICmd`, you might want to call `SCSIRead` with a TIB that reads four bytes (typically a header). After reading the field (in the four-byte header) that tells how many remaining bytes are available, you might call `SCSIRead` again with a TIB to read the remaining bytes. The problem is that the first byte of the second `SCSIRead` data will be lost because of the way the SCSI Manager handles reads in pseudo-DMA mode. The work-around is to issue two separate SCSI commands: the first to read only the four-byte header, the second to read the four-byte header plus the remaining bytes. We recommend that you **not** use a clever TIB that contains two data transfers, the second of which gets the transfer length from the first transfer's received data (the header). These two step TIBs will not work in the future. This bug will probably not be fixed.

- On read operations, some devices may be slow in deasserting `REQ` after sending the last byte to the CPU. The current SCSI Manager (all machines) will return to the caller without waiting for `REQ` to be deasserted. Usually the next call that the driver would make is `SCSIComplete`. On the Macintosh SE and II, the `SCSIComplete` call will check the bus to be sure that it is in `Status` phase. If not, the SCSI Manager will return a new error code that indicates the bus was in Data In/Data Out phase when `SCSIComplete` was called. The combination of the speed of the Macintosh II and a

slow peripheral can cause `SCSIComplete` to detect that the bus is still in Data In phase before the peripheral has finally changed the bus to `Status` phase. This results in a false error being passed back by `SCSIComplete`.

- The `scComp` (compare) TIB opcode does not work in System 4.1 on the Macintosh Plus only. It returns an error code of 4 (bad parameters). This has been fixed in System 4.2.

Other SCSI Manager Issues

- At least one third-party SCSI peripheral driver used to issue SCSI commands from a VBL task. It didn't check to see if the bus was in the free state before sending the command! This is guaranteed to wipe out any other SCSI command that may have been in progress, since the SCSI Manager on the Macintosh Plus does not mask out (or use) interrupts.

  We strongly recommend that you avoid calling the SCSI Manager from interrupt handlers (such as VBL tasks). If you must send SCSI commands from a VBL task (like for a removable media system), do a `SCSIStat` call first to see if the bus is currently busy. If it's free (`BSY` is not asserted), then it's probably safe; otherwise the VBL task should not send the command. Note that you can't call `SCSIStat` before the System file fixes are in place. Since SCSI operations during VBL are not guaranteed, you should check all errors from SCSI Manager calls.

- A new SCSI Manager call will be added in the future. This will be a high-level call; it will have some kind of parameter block in which you give a pointer to a command buffer, a pointer to your TIB, a pointer to a sense data buffer (in case something goes wrong, the SCSI Manager will automatically read the sense bytes into the buffer for you), and a few other fields. The SCSI Manager will take care of arbitration, selection, sending the command, interpreting the TIB for the data transfer, and getting the status and message bytes (and the sense bytes, if there was an error). It should make SCSI device drivers much easier to write, since the driver will no longer have to worry about unexpected phase changes, getting the sense bytes, and so on. In the future, this will be the recommended way to use the SCSI Manager.

- The SCSI Manager (all machines) does not currently support interrupt-driven (asynchronous) operations. The Macintosh Plus can never support it since there is no interrupt capability, although a polled scheme may be implemented by the SCSI Manager. The Macintosh SE has a maskable interrupt for `IRQ`, and the Macintosh II has maskable interrupts for both `IRQ` and `DRQ`. Apple is working on an implementation of the SCSI Manager that will support asynchronous operations on the Macintosh II and probably on the SE as well. Because the interrupt hardware will interact adversely with any asynchronous schemes that are polled, it is strongly recommended that third parties do not attempt asynchronous operations until the new SCSI Manager is released. Apple will not attempt to be compatible with any products that bypass some or all of the SCSI Manager. In order to implement software-based (polled) asynchronous operations it is necessary to bypass the SCSI Manager.

The SCSI Manager section of the alpha draft of *Inside Macintosh* volume V documented the Disconnect and Reselect routines which were intended to be used for asynchronous I/O. Those routines cannot be used. Those routines have been removed from the manual. Any software that uses those routines will have to be revised when the SCSI Manager becomes interrupt-driven. Drivers which send SCSI commands from VBL tasks may also have to be modified.

Hardware in the SCSI

There is some confusion on how many terminators can be used on the bus, and the best way to use them. There can be no more than two terminators on the bus. If you have more than one SCSI drive you must have two terminators. If you only have one drive, you should use a single terminator. If you have more than one drive, the two terminators should be on opposite ends of the chain. The idea is to terminate both ends of the wire that goes through all of the devices. One terminator should be on the end of the system cable that comes out of the Macintosh. The other terminator would be on the very end of the last device on the chain. If you have an SE or II with an internal hard disk, there is already one terminator on the front of the chain, inside the computer.

On the Macintosh SE and II, there is additional hardware support for the SCSI bus transfers in pseudo-DMA mode. The hardware makes it possible to handshake the data in Blind mode so that the Blind mode is safe for all transfers. On the Macintosh Plus, the Blind transfers are heavily timing dependent and can overrun or underrun during the transfer with no error generated. Assuring that Blind mode is safe on the Macintosh Plus depends upon the peripheral being used. On the SE and II, the transfer is hardware assisted to prevent overruns or underruns.

Changes in SCSI for SE and II

The changes made to the SCSI Manager found in the Macintosh SE and Macintosh II are primarily bug fixes. No new functionality was added. The newer SCSI Manager is more robust and has more error checking. Since the Macintosh Plus SCSI Manager only did limited error checking, it is possible to have code that would function (with bugs) on the Macintosh Plus, but will not work correctly on the SE or II. The Macintosh Plus could mask some bugs in the caller by not checking errors. An example of this is sending or receiving the wrong number of bytes in a blind transfer. On the Macintosh Plus, no error would be generated since there was no way to be sure how many bytes were sent or received. On the SE and II, if the wrong number of bytes are transferred an error will be returned to the caller. The exact timing of transfers has changed on the SE and II as well, since the computers run at different speeds. Devices that are unwittingly dependent upon specific timing in transfers may have problems on the newer computers. To find problems of this sort it is usually only necessary to examine the error codes that are passed back by the SCSI Manager routines. The error codes will generally point out where the updated SCSI Manager found errors.

To report other bugs or make suggestions

Please send additional bug reports and suggestions to us at the address in Technical Note #0. Let us know what SCSI controller you're using in your peripheral, and whether you've had any particularly good or bad experiences with it. We'll add to this note as more information becomes available.

#97: PrSetError Problem

Written by: Mark Baumwell
Updated:

November 15, 1986
March 1, 1988

This note formerly described a problem in Lisa Pascal glue for the `PrSetError` routine. The glue in MPW (and most, if not all, third party compilers) does not have this problem.

#98: Short-Circuit Booleans in Lisa Pascal

Written by: Mark Baumwell
Updated:

November 15, 1986
March 1, 1988

This note formerly described problems with the Lisa Pascal compiler. These problems have been fixed in the MPW Pascal compiler.

#99: Standard File Bug in System 3.2

See also: The Standard File Package

Written by: Jim Friedlander
Updated:

November 15, 1986
March 1, 1988

This note formerly described a bug in Standard File in System 3.2. This bug has been fixed in more recent Systems.

#100: Compatibility with Large-Screen Displays

See also: Technical Note #2—Macintosh Compatibility Guidelines

Written by: Bryan Stearns November 15, 1986
Updated: March 1, 1988

A number of third-party developers have announced large-screen display peripherals for Macintosh. One of them, Radius Inc., has issued a set of guidelines for developers who wish to remain compatible with their Radius FPD; unfortunately, one of their recommendations can cause system crashes. This note suggests a more correct approach.

On the first page of the appendix to their guidelines, "How to be FPD Aware," Radius recommends the following:

> "First, to detect the presence of a Radius FPD, you should check address `$C00008`..."

Unfortunately, this assumes that you're running on a Macintosh or Macintosh Plus; this test will not work on Macintosh XL, nor on a Macintosh II. Since these displays weren't designed to work with systems other than Macintosh and Macintosh Plus, you should make sure you're running on one of these systems before addressing I/O locations (such as those for an add-on display).

Before testing for the presence of any large-screen display, you should first check the machine ID; it's the byte located at `(ROMBASE)+8` (that is, take the long integer at the low-memory location `ROMBASE` [`$2AE`], and add 8 to get the address of the machine ID byte. On a Macintosh or Macintosh Plus, this address will work out to be `$400008`; however, use the low-memory location, to be compatible with future systems that may have the ROM at a different address!).

The machine ID byte will be `$00` for all current Macintosh systems. If the value isn't `$00`, you can assume that no large-screen display is present, but don't forget to follow Technical Note #2's guidelines for screen size independence!

Note: If you are a developer of an add-on large-screen display, we'd be happy to review your guidelines for developers in advance of distribution; please send them to us at the address for comments in Technical Note #0. Future versions of this note may recommend general guidelines for dealing with add-on large-screen displays.

#101: CreateResFile and the Poor Man's Search Path

See also: The File Manager
The Resource Manager
Technical Note #77—HFS Ruminations

Written by: Jim Friedlander
Updated:

January 12, 1987
March 1, 1988

`CreateResFile` checks to see if a resource file with a given name exists, and if it does, returns a `dupFNErr` (–48) error. Unfortunately, to do this check, `CreateResFile` uses a call that follows the Poor Man's Search Path (PMSP).

`CreateResFile` checks to see if a resource file with a given name exists, and if it does, returns a `dupFNErr` (–48) error. Unfortunately, to do the check, `CreateResFile` calls `PBOpenRF`, which uses the Poor Man's Search Path (PMSP). For example, if we have a resource file in the System folder named '`MyFile`' (and no file with that name in the current directory) and we call `CreateResFile('MyFile'),ResError` will return a `dupFNErr`, since `PBOpenRF` will search the current directory first, then search the blessed folder on the same volume. This makes it impossible to use `CreateResFile` to create the resource file '`MyFile`' in the current directory if a file with the same name already exists in a directory that's in the PMSP.

To make sure that `CreateResFile` will create a resource file in the current directory whether or not a resource file with the same name already exists further down the PMSP, call `_Create` (`PBCreate` or `Create`) before calling `CreateResFile`:

```
err := Create('MyFile',0,myCreator,myType);
            {0 for VRefNum means current volume/directory}
CreateResFile('MyFile');
err := ResError; {check for error}
```

In MPW C:

```
err = Create("\pMyFile",0,myCreator,myType);
CreateResFile("\pMyFile");
err = ResError();
```

This works because `_Create` does **not** use the PMSP. If we already have '`MyFile`' in the current directory, `_Create` will fail with a `dupFNErr`, then, if '`MyFile`' has an empty resource fork, `CreateResFile` will write a resource map, otherwise, `CreateResFile` will return `dupFNErr`. If there is no file named '`MyFile`' in the current directory, `_Create` will create one and then `CreateResFile` will write the resource map.
Notice that we are intentionally ignoring the error from `_Create`, since we are calling it

only to assure that a file named 'MyFile' does exist in the current directory.

Please note that SFPutFile does **not** use the PMSP, but that FSDelete does. SFPutFile returns the vRefNum/WDRefNum of the volume/folder that the user selected. If your program deletes a resource file before creating one with the same name based on information returned from SFPutFile, you can use the following strategy to avoid deleting the wrong file, that is, a file that is not in the directory specified by the vRefNum/WDRefNum returned by SFPutFile, but in some other directory in the PMSP:

```
VAR
   wher   : Point;
   reply  : SFReply;
   err    : OSErr;
   oldVol : Integer;

...

   wher.h := 80; wher.v := 90;
   SFPutFile(wher,'','',NIL,reply);
   IF reply.good THEN BEGIN
       err := GetVol(NIL,oldVol); {So we can restore it later}
       err := SetVol(NIL,reply.vRefNum);{for the CreateResFile call}

   {Now for the Create/CreateResFile calls to create a resource file that
    we know is in the current directory}

       err := Create(reply.fName,reply.vRefNum,myCreator,myType);
       CreateResFile(reply.fName); {we'll use the ResError from this ...}

       CASE ResError OF
          noErr:{the create succeeded, go ahead and work with the new
                  resource file -- NOTE: at this point, we don't know
                  what's in the data fork of the file!!} ;
          dupFNErr: BEGIN {duplicate file name error}
              {the file already existed, so, let's delete it. We're now
               sure that we're deleting the file in the current directory}

                 err:= FSDelete(reply.fName,reply.vRefNum);

              {now that we've deleted the file, let's create the new one,
               again, we know this will be in the current directory}

                 err:= Create(reply.fName,reply.vRefNum,myCreator,myType);
                 CreateResFile(reply.fName);
          END; {CASE dupFNErr}
          OTHERWISE     {handle other errors} ;
       END;             {Case ResError}
       err := SetVol(NIL,oldVol);{restore the default directory}
   END;                 {If reply.good}
...
```

In MPW C:

```
Point         wher;
SFReply       reply;
OSErr         err;
short         oldVol;


wher.h = 80; wher.v = 90;
SFPutFile(wher,"","",nil,&reply);
if (reply.good )
{
    err = GetVol(nil,&oldVol);
    /*So we can restore it later*/
    err = SetVol(nil,reply.vRefNum);/*for the CreateResFile call*/

    /*Now for the Create/CreateResFile calls to create a resource file
    that we know is in the current directory*/

    err = Create(&reply.fName,reply.vRefNum,myCreator,myType);
    CreateResFile(&reply.fName);
    /*we'll use the ResError from this ...*/

    switch (ResError())
    {
       case noErr:;/*the create succeeded, go ahead and work with the
                    new resource file -- NOTE: at this point, we don't
                    know what's in the data fork of the file!!*/
            break; /* case noErr*/
       case dupFNErr: /*duplicate file name error*/
                   /*the file already existed, so, let's delete it.
                   We're now sure that we're deleting the file in the
                   current directory*/

                   err= FSDelete(&reply.fName,reply.vRefNum);

                   /*now that we've deleted the file, let's create the
                   new one, again, we know this will be in the current
                   directory*/

                   err= Create(&reply.fName,reply.vRefNum,
                                          myCreator,myType);
                   CreateResFile(&reply.fName);
           break; /*case dupFNErr*/
       default:;     /*handle other errors*/
    }  /* switch */
    err = SetVol(nil,oldVol);/*restore the default directory*/
}                  /*if reply.good*/
```

Note: `OpenResFile` uses the PMSP too, so you may have to adopt similar strategies to make sure that you are opening the desired resource file and not some other file further down the PMSP. This is normally not a problem if you use `SFGetFile`, since `SFGetFile` does not use the PMSP, in fact, `SFGetFile` does not open or close files, so it doesn't run into this problem.

#102: HFS Elucidations

See also: The File Manager
Technical Note #77—HFS Ruminations

Written by: Bryan "Bo3b" Johnson
Updated:

January 12, 1987
March 1, 1988

This technical note will describe a few problems that can occur while using HFS. It will also describe ways to avoid these problems.

This technical note will discuss the following problems:

1) It is very important to be careful about how files are opened and closed. There must be no more than one close for every open.

2) Don't use Driver names, like `.Bout`, `.Print` or `.Sony`, in place of file names or the file system will become confused.

3) Be aware of the `ioFlVersNum` byte in all file calls. A number of pieces of the Macintosh system do not use, and may in fact ignore, files created with non-zero `ioFlVersNums`.

Each of these can lead to strange occurrences, as well as problems for the users. Doing any or all of these marginally illegal operations will not necessarily lead to a System Error. In some cases the confusion generated may be worse than a System Error.

One Close is always enough

If a file is closed twice, it is possible to corrupt the file system on a disk. If a program has been creating unreadable disks, this may be the cause.

One aspect of the file system that is not well documented is how it allocates access paths to files that are currently open. As a result of this, it is possible to get a rather cavalier attitude about opening and closing files. This discussion will explain why it is necessary to be very careful about opening and closing files.

When the File Manager receives an `Open` call, it will look at the parameters passed in the parameter block and create a new access path for the file that is being opened. The access path is how the File Manager keeps track of where to send data that is written, and where to get data that is read from that file. An access path is nothing more than: 1) a buffer that the file system uses to read and write data, and 2) a File Control Block that

describes how the file is stored on a disk.

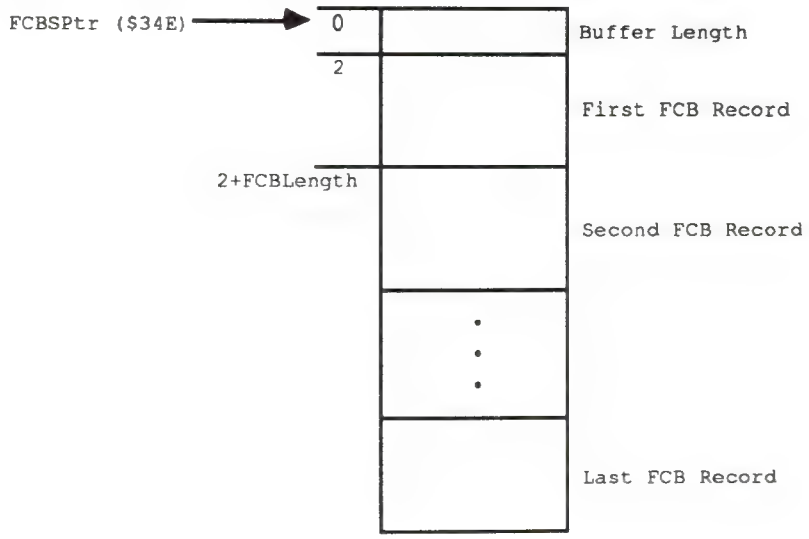
A call like:

```
ErrStuff := FSOpen ('FirstFile', theVRefNum, FirstRefNum);
```

will create the access path as a buffer and a File Control Block (FCB) in the FCB queue.

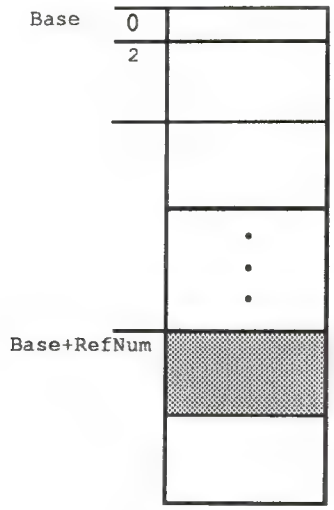
Note: The following information is here for illustrative purposes only; dependence on it may cause compatibility problems with future system software.

The structure of the queue can be visualized as:



where `FCBSPtr` is a low-memory global (at `$34E`) that holds the address of a nonrelocatable block. That block is the File Control Block buffer, and is composed of the two byte header which gives the length of the block, followed by the FCB records themselves. The records are of fixed length, and give detailed information about an open file. As depicted, any given record can be found by adding the length of the previous FCB records to the start of the block, adding 2 for the two byte header; giving an offset to the record itself. The size of the block, and hence the number of files that can be open at any given time, is determined at startup time. The call to open '`FirstFile`' above will pass back the File Reference Number to that file in `FirstRefNum`. This is the number that will be used to access that file from that point on. The File Manager passes back an offset into the FCB queue as the `RefNum`. This offset is the number of bytes past the beginning of the queue to that FCB record in the queue. That FCB record will describe the file that was opened. An example of a number that might get passed back as a `RefNum` is `$1D8`. That also means that the FCB record is `$1D8` bytes into the FCB block.

A visual example of a record being in use, and how the RefNum is related is:

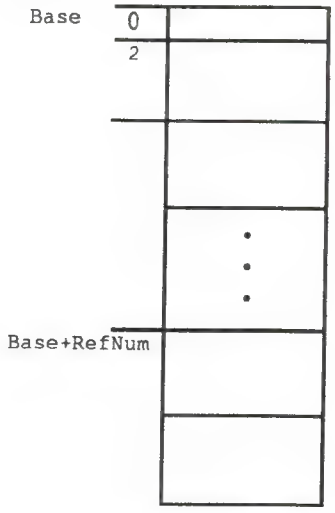


Base is merely the address of the nonrelocatable block that is the FCB buffer. FCBSPtr points to it. The RefNum (a number like $1D8) is added to Base, to give an address in the block. That address is what the file system will use to read and write to an open file, which is why you are required to pass the RefNum to the PBRead and PBWrite calls.

Since that RefNum is merely an offset into the queue, let's step through a dangerous imaginary sequence and see what happens to a given record in the FCB Buffer. Here's the sequence we will step through:

```
ErrStuff := FSOpen ('FirstFile', theVRefNum, FirstRefNum);
ErrStuff := FSClose ( FirstRefNum );
ErrStuff := FSOpen ('SecondFile', theVRefNum, SecondRefNum);
ErrStuff := FSClose ( FirstRefNum ); {the wrong file gets closed!!!}
{the above line will close 'SecondFile', not 'FirstFile', which is already
 closed}
```

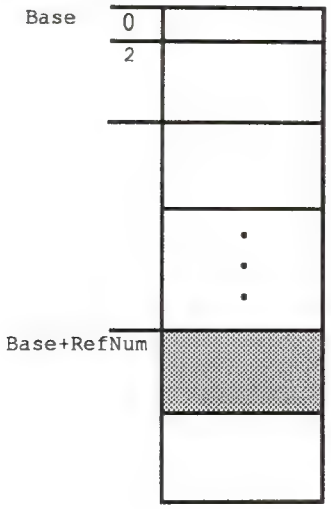
Before any operations:
the record at $1D8 is not used.



After the call:

```
ErrStuff := FSOpen ('FirstFile', theVRefNum, FirstRefNum);
```

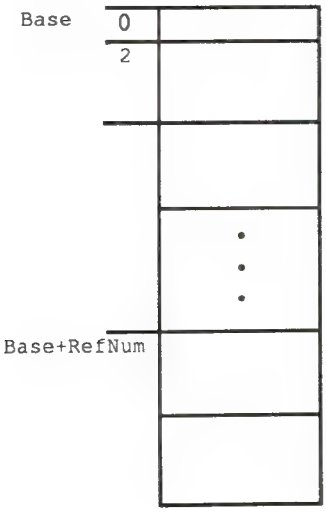
FirstRefNum = $1D8 and the record is in use.



After the call:

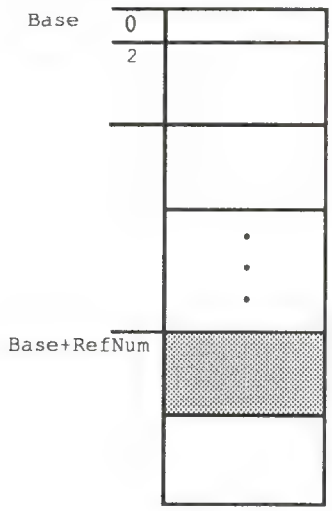
```
ErrStuff := FSClose (FirstRefNum);
```

FirstRefNum is still equal to $1D8, but the FCB record is unused.



After the call:

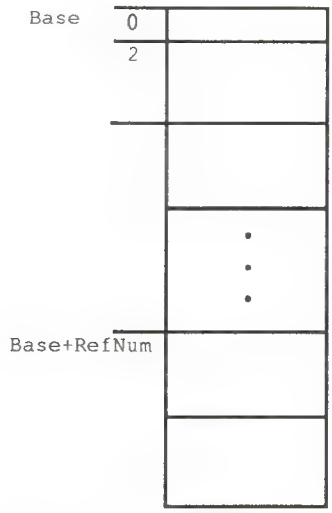
```
ErrStuff := FSOpen ('SecondFile', theVRefNum, SecondRefNum);
SecondRefNum = $1D8, FirstRefNum = $1D8, and the record is reused.
```



After the call:

```
ErrStuff := FSClose (FirstRefNum);
The FirstRefNum = $1D8, SecondRefNum = $1D8,
```

the queue element is cleared. This happens, even though `FirstFile` was already closed. Actually, `SecondFile` was closed:



Note that the second close is using the old `RefNum`. The second close will still close a file, and in fact will return `noErr` as its result. Any subsequent accesses to the `SecondRefNum` will return an error, since the file '`SecondFile`' was closed. The File Control Blocks are reused, and since they are just offsets, it is possible to get the same file `RefNum` back for two different files. In this case, `FirstRefNum` = `SecondRefNum` since '`FirstFile`' was closed before opening '`SecondFile`' and the same FCB record was reused for '`SecondFile`'.

There are worse cases than this, however. As an example, think of what can happen if a program were to close a file, then the user inserted an HFS disk. The FCB could be reused for the Catalog File on that HFS disk. If the program had a generic error handler that closed all of its files, it could inadvertently close "its" file again. If it thought "its" file was still open it would do the close, which could close the Catalog file on the HFS disk. This is catastrophic for the disk since the file could easily be closed in an inconsistent state. The result is a bad disk that needs to be reformatted.

There are any number of nasty cases that can arise if a file is closed twice, reusing an old `RefNum`. A common programming practice is to have an error handler or cleanup routine that goes through the files that a program creates and closes them all, even if some may already be closed. If an FCB element was not reused, the `Close` will return the expected `fnOpnErr`. If the FCB had been reused, then the `Close` could be closing the wrong file. This can be very dangerous, particularly for all those paranoid hard disk users.

How to avoid the problem:

A very simple technique is to merely clear the `RefNum` after each close. If the variable that the program uses is cleared after each close, then there is no way of reusing a `RefNum` in the program. An example of this technique would be:

```
ErrStuff := FSOpen ('FirstFile', theVRefNum, FirstRefNum);
ErrStuff := FSClose (FirstRefNum);
FirstRefNum := 0; { We just closed it, so clear our refnum }
ErrStuff := FSOpen ('SecondFile', theVRefNum, SecondRefNum);
ErrStuff := FSClose (FirstRefNum); { returns an error }
```

This makes the second `Close` pass back an error. In this case, the second close will try to close `RefNum` = 0, which will pass back a `fnOpnErr` and do no damage. **Note:** Be sure to use 0, which will never be a valid `RefNum`, since the first FCB entry is beyond the FCB queue length word. Don't confuse this with the 0 that the Resource Manager uses to represent the System file.

Thus, if an error handler were cleaning up possibly open files, it could blithely close all the files it knew about, since it would legitimately get an error back on files that are already closed. This is not done automatically, however. The programmer must be careful about the opening and closing of files. The problem can get quite complex if an error is received halfway through opening a sequence of ten files, for example. By merely clearing the `RefNum` that is stored after each close, it is possible to avoid the complexities of trying to track which files are open and which are closed.

This .file name looks outrageous.

There is a potential conflict between file names and driver names. If a file name is named something like `.Bout`, `.Print` or `.Sony`, then the file system will open the driver instead of the file. Drivers have priority on the 128K ROMs, and will always be opened before a file of the same name. This may mean that an application will get an error back

when opening these types of files, or worse, it will get back a driver `RefNum` from the call. What the application thought was a file open call was actually a driver open call. If the program uses that access path as a file `RefNum`, it is possible to get all kinds of strange things to happen. For example, if `.Sony` is opened, the Sony driver's `RefNum` would be passed back, instead of a file `RefNum`. If the application does a `Write` call using that `RefNum`, it will actually be a driver call, using whatever parameters happen to be in the parameter block. Disks may be searching for new life after this type of operation. If a program creates files, it should not allow a file to be created whose name begins with '.'.

This file's not my type.

This has been discussed in other places, but another aspect of the File Manager that can cause confusion is the `ioFlVersNum` byte that is passed to the low-level File Manager calls. This is called `ioFileType` from Assembly, and should not be confused with `ioFVersNum`. This byte must be set to zero for normal Macintosh files. There are a number of parts of the system that will not deal correctly with files that have the wrong versions: the Standard File package will not display any file with a non-zero `ioFlVersNum`; the Segment Loader and Resource Manager cannot open files that have non-zero `ioFlVersNums`. It is not sufficient to ignore this byte when a file is created. The byte must be cleared in order to avoid this type of problem. Strictly speaking, it is not a problem unless a file is being created on an MFS disk. The current system will easily allow the user to access 400K disks however, so it is better to be safe than confused.

#103: Using MaxApplZone and MoveHHi from Assembly Language

See also: Using Assembly Language
The Memory Manager
Technical Note #129—SysEnvirons

Written by: Bryan "Bo3b" Johnson
Updated:

January 12, 1987
March 1, 1988

When calling `MaxApplZone` and `MoveHHi` from assembly language, be sure to get the correct code.

`MaxApplZone` and `MoveHHi` were marked [Not in ROM] in *Inside Macintosh, Volumes I-III* . They are ROM calls in the 128K ROM. Since they are not in the 64K ROM, if you want your program to work on 64K ROM routines it is necessary to call the routines by a `JSR` to a glue (library) routine instead of using the actual trap macro. The glue calls the ROM routines if they are available, or executes its copy of them (linked into your program) if not.

How to do it:

Whenever you need to use these calls, just call the library routine. It will check `ROM85` to determine which ROMs are running, and do the appropriate thing.

For MDS, include the `Memory.Rel` library in your link file and use:

```
XREF  MoveHHi     ; we need to use this 'ROM' routine
...
JSR  MoveHHi      ; jump to the glue routine that will check ROM85 for us
```

For MPW link with `Interface.o` and use:

```
IMPORT  MoveHHi    ; we need to use this
...
JSR  MoveHHi       ; jump to the glue routine that will check ROM85 for us
```

Avoid calling `_MaxApplZone` or `_MoveHHi` directly if you want your software to work on the 64K ROMs, since that will assemble to an actual trap, not to a `JSR` to the library.

If your program is going to be run **only** on machines with the 128K ROM or newer, you can call the traps directly. Be sure to check for the 64K ROMs, and report an error to the user. You can check for old ROMs using the `SysEnvirons` trap as described in Technical Note #129.

#104: MPW: Accessing Globals From Assembly Language

See also: MPW Reference Manual

Written by: Jim Friedlander
Updated:

January 12, 1987
March 1, 1988

This technical note demonstrates how to access MPW Pascal and MPW C globals from the MPW Assembler.

To allow access of MPW Pascal globals from the MPW Assembler, you need to identify the variables that you wish to access as external. To do this, use the {$Z+} compiler option. Using the {$Z+} option can substantially increase the size of the object file due to the additional symbol information (no additional code is generated and the symbol information is stripped by the linker). If you are concerned about object file size, you can "bracket" the variables you wish to access as external variables with {$Z+} and {$Z-}. Here's a trivial example:

Pascal Source

```
PROGRAM MyPascal;
USES
   MemTypes,QuickDraw,OSIntf,ToolIntf;

VAR
   myWRect: Rect;
{$Z+} {make the following external}
   myInt: Integer;
{$Z-} {make the following local to this file (not lexically local)}
   err: Integer;

PROCEDURE MyAsm; EXTERNAL; {routine doubles the value of myInt}

BEGIN {PROGRAM}
   myInt:= 5;
   MyAsm; {call the routine, myInt will be 10 now}
   writeln('The value of myInt after calling myAsm is ', myInt:1);
END. {PROGRAM}
```

Assembly Source for Pascal

```
        CASE    OFF             ;treat upper and lower case identically
MyAsm   PROC    EXPORT          ;CASE OFF is the assembler's default
        IMPORT myInt:DATA       ;we need :DATA, the assembler assumes CODE
        ASL.W  #1,myInt         ;multiply by two
        RTS                     ;all done with this extensive routine, whew!
```

```
        END
```

The variable `myInt` is accessible from assembler. Neither `myWRect` nor `err` are accessible. If you try to access `myWRect`, for example, from assembler, you will get the following linker error:

```
### Link: Error   Undefined entry name:   MYWRECT.
```

C Source

In an MPW C program, one need only make sure that `MyAsm` is declared as an external function, that `myInt` is a global variable (capitalizations must match) and that the `CASE ON` directive is used in the Assembler:

```
#include <types.h>
#include <quickdraw.h>
#include <fonts.h>
#include <windows.h>
#include <events.h>
#include <textedit.h>
#include <dialogs.h>
#include <stdio.h>

extern MyAsm();     /* assembly routine that doubles the value of myInt */
short myInt;       /* we'll change the value of this variable from MyAsm */

main()
{
WindowPtr MyWindow;
Rect myWRect;

myInt = 5;
MyAsm();
printf(" The value of myInt after calling myAsm is %d\n",myInt);
} /*main*/
```

Assembly source for C

```
        CASE    ON              ;treat upper and lower case distinct
MyAsm   PROC    EXPORT          ;this is how C treats upper and lower case
        IMPORT myInt:DATA       ;we need :DATA, the assembler assumes CODE
        ASL.W  #1,myInt         ;multiply by two
        RTS                     ;all done with this extensive routine, whew!
        END
```

#105: MPW Object Pascal Without MacApp

See also: Technical Note #93—{$LOAD};_DataInit;%_MethTables

Written by: Rick Blair January 12, 1987
Updated: March 1, 1988

Object Pascal must have a CODE segment named `%_MethTables` in order to access object methods. In MacApp this is taken care of "behind the scenes" so you don't have to worry about it . However, if you are doing a straight Object Pascal program, you must make sure that `%_MethTables` is around when you need it. If it's unloaded when you call a method, your Macintosh will begin executing wild noncode and die a gruesome and horrible death.

The MPW Pascal compiler must see some declaration of an object in order to produce a reference to the magic segment. You can achieve this cheaply by simply including `ObjIntf.p` in your `Uses` declaration. This must be in the main program, by the way. The compiler will produce a call to `%_InitObj` which is in `%_MethTables`.

If you're a more adventurous soul, you can call `%_InitObj` explicitly from the initialization section of your main program (you must use the `{$%+}` compiler directive to allow the use of "`%`" in identifiers). This will load the `%_MethTables` segment. See Technical Note #93 for ideas about locking down segments that are needed forever without fragmenting the heap.

#106: The Real Story: VCBs and Drive Numbers

See also: The File Manager
Technical Note #36—Drive Queue Element Format

Written by: Rick Blair January 12, 1987
Updated: March 1, 1988

The top of page IV-178 in The File Manager chapter of *Inside Macintosh* in attempts to explain the behavior of two fields in a volume control block when the corresponding disk is offline or ejected. Due to the fact that a little bit is left unsaid, this paragraph is rather misleading. The two fields in question are `vcbDrvNum` and `vcbDRefNum` (referred to as `ioVDrvInfo` and `ioVDRefNum` in C and Pascal). `PBHGetVInfo` can be used to access these fields.

Offline

When a mounted volume is placed offline, `vcbDrvNum` is cleared **and** `vcbDRefNum` is set to the two's complement of the drive number. Since drive numbers are assigned positive values (starting with one), this will be a negative number. If `vcbDrvNum` is zero **and** `vcbDRefNum` is negative, you know that the volume is offline.

Ejected

When a volume is ejected, `vcbDrvNum` is cleared and `vcbDRefNum` is set to the positive drive number. If `vcbDrvNum` is zero and `vcbDRefNum` is positive, you know that the volume is ejected. Ejection implies being offline. There is no such thing as "premature ejection".

Summary

	online	offline	ejected
`vcbDrvNum`	`>0 (DrvNum)`	0	0
`vcbDRefNum`	`<0 (DRefNum)`	`<0 (-DrvNum)`	`>0 (DrvNum)`

Please refrain from assuming anything about a VCB queue element beyond what is documented in *Inside Macintosh*, and don't expect it to always be 178 bytes in size. It grew when we went from MFS to HFS, and it may grow again. It's safest to use calls like `PBHGetVInfo` to get the information that you need.

#107: Nulls in Filenames

See also: The File Manager

Written by: Rick Blair
Updated:

March 2, 1987
March 1, 1988

Some applications (loosely speaking so as to include Desk Accessories, INITs, and what-have-you) generate or rename special files on the fly so that they are not explicitly named by the user via `SFPutFile`. Since the Macintosh file system is very liberal about filenames and only excludes colons from the list of acceptable characters, this can lead to some difficulties, both for the end user and for writers of other programs which may see these files.

Other programs which might be backing up your disk or something similar may get confused. A program written in C will think it has found the end of a string when it hits a null (ASCII code 0) character, so nulls in filenames are especially risky.

As a rule, filenames should only include characters which the user can see and edit. The only reasonable exception might be invisible files, but it can be argued that they are of dubious value anyway. You can argue "but what about my help file, I don't want it renamed" but we already have what we think is the best approach for that situation. If you can't find a configuration or other file because the user has renamed or moved it, then call `SFGetFile` and let the user find it. If the user cancels, and you can't run without the file, then quit with an appropriate message.

Please consider carefully before you put non-displaying characters in filenames!

#108: _AddDrive, _DrvrInstall, and _DrvrRemove

See also: Technical Note #36, Drive Queue Elements
SCSI Development Package (APDA)

Written by: Jim Friedlander
Revised by: Pete Helme

March 2, 1987
December 1988

`_AddDrive`, `_DrvrInstall`, and `_DrvrRemove` are used in the sample SCSI driver in the SCSI Development Package, which is available from APDA. This Technical Note documents the parameters for these calls.
Changes since March 1, 1988: Updated the `_DrvrInstall` text to reflect the use of register `A0`, which should contain a pointer to the driver when called. Also added simple glue code for `_DrvrInstall` and `_DrvrRemove` since none is available in the MPW interfaces.

_AddDrive

`_AddDrive` adds a drive to the drive queue, and is discussed in more detail in Technical Note #36, Drive Queue Elements:

```
FUNCTION AddDrive(DQE:DrvQE1;driveNum,refNum:INTEGER):OSErr;
```

`A0` (input)	→	pointer to `DQE`
`D0` high word(input)	→	drive number
`D0` low word(input)	→	driver `RefNum`
`D0` (output)	←	error code
		`noErr` (always returned)

_DrvrInstall

`_DrvrInstall` is used to install a driver. A `DCE` for the driver is created and its handle entered into the specified Unit Table position (–1 through –64). If the unit number is –4 through –9, the corresponding ROM-based driver will be replaced:

```
FUNCTION DrvrInstall(drvrHandle:Handle; refNum: INTEGER): OSErr;
```

`A0` (input)	→	pointer to driver
`D0` (input)	→	driver `RefNum` (–1 through –64)
`D0` (output)	←	error code
		`noErr`
		`badUnitErr`

_DrvrRemove

_DrvrRemove is used to remove a driver. A RAM-based driver is purged from the system heap (using _ReleaseResource). Memory for the DCE is disposed:

```
FUNCTION DrvrRemove(refNum: INTEGER):OSErr;
```

D0 (input)	→	Driver RefNum
D0 (output)	←	error code
		noErr
		qErr

Interfaces

Through a sequence of cataclysmic events, the glue code for _DrvrInstall and _DrvrRemove was never actually added to the MPW interfaces (i.e., "We forgot."), so we will include simple glue here at no extra expense to you.

It would be advisable to first lock the handle to your driver with _HLock before making either of these calls since memory may be moved.

```
;--------------------------------------------------------------
; FUNCTION DRVRInstall(drvrHandle:Handle; refNum:INTEGER):OSErr;
;--------------------------------------------------------------

DRVRInstall    PROC    EXPORT
       MOVEA.L         (SP)+, A1       ; pop return address
       MOVE.W          (SP)+, D0       ; driver reference number
       MOVEA.L         (SP)+, A0       ; handle to driver
       MOVEA.L         (A0), A0        ; pointer to driver
       _DrvrInstall                    ; $A03D
       MOVE.W          D0, (SP)        ; get error
       JMP             (A1)            ; & split
       ENDPPROC


;--------------------------------------------------------------
; FUNCTION DRVRRemove(refNum:INTEGER):OSErr;
;--------------------------------------------------------------

DRVRRemove     PROC    EXPORT
       MOVEA.L         (SP)+, A1       ; pop return address
       MOVE.W          (SP)+, D0       ; driver reference number
       _DrvrRemove                     ; $A03E
       MOVE.W          D0, (SP)        ; get error
       JMP             (A1)            ; & split
       ENDPPROC
```

#109: Bug in MPW 1.0 Language Libraries

See also: MPW Reference Manual

Written by: Scott Knaster
Updated:

March 2, 1987
March 1, 1988

This note formerly described a problem in the language libraries for MPW 1.0. This bug is fixed in MPW 1.0.2, available from APDA.



#110: MPW: Writing Stand-Alone Code

Revised by: Keith Rollin
Written by: Jim Friedlander

August 1990
March 1987

This Technical Note formerly discussed using MPW Pascal and C to write stand-alone code, such as `'WDEF'`, `'LDEF'`, `'INIT'`, and `'FKEY'` resources.
Changes since February 1990: Merged the contents of this Note into Technical Note #256, Stand-Alone Code, *ad nauseam.*

This Note formerly discussed using MPW Pascal and C to write stand-alone code. This information has been expanded and is now contained in Technical Note #256, Stand-Alone Code, *ad nauseam.*

#111: MoveHHi and SetResPurge

See also: The Memory Manager
The Resource Manager

Written by: Jim Friedlander March 2, 1987
Updated: March 1, 1988

`SetResPurge(TRUE)` is called to make the Memory Manager call the Resource Manager before purging a block specified by a handle. If the handle is a handle to a resource, and its `resChanged` bit is set, the resource data will be written out (using `WriteResource`).

When `MoveHHi` is called, even though the handle's block is not actually being purged, the resource data specified by the handle will be written out. An application can prevent this by calling `SetResPurge(FALSE)` before calling `MoveHHi` (and then calling `SetResPurge(TRUE)` after the `MoveHHi` call).

#112: FindDItem

See also: The Dialog Manager

Written by: Rick Blair
Updated:

March 2, 1987
March 1, 1988

`FindDItem` is a potentially useful call which returns the number of a dialog item given a point in local coordinates and a dialog handle. It returns an item number of –1 if no item's rectangle overlaps the point. This is all well and good, except you don't get back quite what you would expect.

The item number returned is zero-based, so you have to add one to the result:

```
theitem := FindDItem(theDialog, thePoint) + 1;
```

#113: Boot Blocks

See also: The Segment Loader

Written by: Bo3b Johnson March 2, 1987
Updated: March 1, 1988

There are two undocumented features of the Boot Blocks. This note will describe how they currently work.

Warning: The format and functionality of the Boot Blocks will change in the future; dependence on this information may cause your program to fail on future hardware or with future System software.

The first two sectors of a bootable Macintosh disk are used to store information on how to start up the computer. The blocks contain various parameters that the system uses to startup such as the name of the system file, the name of the Finder, the first application to run at boot time, the number of events to allow, etc.

Changing System Heap Size

The boot blocks dictate what size the system heap will be after booting. Any common sector editing program will allow you to change the data in the boot blocks. Changing the system heap size is accomplished by changing two parameters in the boot blocks: the long word value at location `$86` in Block 0 indicates the size of the system heap; the word value at location `$6` is the version number of the boot blocks. Changing the version number to be greater than `$14` (`$15` is recommended) tells the ROM to use the value at `$86` for the system heap size, otherwise the value at `$86` is ignored. The `$86` location only applies to computers with more than 128K of RAM.

Secondary Sound and Video Pages

Another occasionally useful feature of the boot blocks is the ability to specify that the secondary sound and video pages be allocated at boot time. This is done before a debugger is loaded, so the debugger will load below the alternate screen. This is useful for debugging software that uses the alternate video page, like page-flipping demos or games. To allocate the second video and sound buffers, change the two bytes starting at location `$8` in the boot blocks. Change the value (normally 0) to a negative number (`$FFFF`) to allocate both video and sound buffers. Change the value to a positive number (`$0001`) to allocate only the secondary sound buffer.

Warning: MacsBug may not work properly if you allocate additional pages for sound and video.

#114: AppleShare and Old Finders

See also: *AppleShare User's Guide*

Written by: Bryan Stearns
Updated:

March 2, 1987
March 1, 1988

A rumor has been spread that if you use a pre-AppleShare Finder on a workstation to access AppleShare volumes, you can bypass AppleShare's "access privilege" mechanisms.

This is not true. Access controls are enforced by the server, **not** by the Finder. If you use an older Finder, you are still prevented (by the server) from gaining access to protected files and folders; however, you will not get the proper user-interface feedback that you would if you were using the correct Finder: for instance, folders on the server will always appear plain white (that is, without the permission feedback you'd normally get), and error messages would not be as explanatory as those from Finders that "know" about AppleShare servers.

#115: Application Configuration with Stationery Pads

See also: The File Manager
Technical Note #116—AppleShare-able Applications
Technical Note #47—Customizing SFGetFile
Technical Note #48—Bundles
"Application Development in a Shared Environment"

Written by: Bryan Stearns March 2, 1987
Updated: March 1, 1988

With the introduction of AppleShare (Apple's file server) there are restrictions on self-modification of application resource files and the placement of configuration files. This note describes one way to get around the necessity for configuration files.

Configuration Files

Some applications need to store information about configuration; others could benefit simply from allowing users to customize default ruler settings, window placement, fonts, etc.

There are applications which store this information as additional resources in the application's resource file; when the user changes the configuration, the application writes to itself to change the saved information.

AppleShare, however, requires that if an application is to be used by more than one user at a time, it must not need write access to itself. This means that the above method of storing configuration information cannot be used. (For more information about making your application sharable, see Technical Note #116.)

Storing configuration in a special configuration file can be a problem; the user must keep the file in the system folder or the application must search for it. This process has design issues of its own.

An alternative to configuration files: Stationery Pads

A basis for one solution to this problem was a user-interface feature of the Lisa Office System architecture. Lisa introduced the concept of "stationery pads", special documents that created copies of themselves to allow users to save a pre-set-up document for future use. On Lisa, this was the way Untitled documents were created.

Your Macintosh application can provide the option of saving a document as a stationery pad, to provide similar functionality. Here's how:

- You'll need to add a checkbox to your `SFPutFile` dialog box (if you don't know how to do this, check out Technical Note #47); if the user checks this box, save the document as you normally would, but use a different file type (the file type of a document is usually set when the document is created, using the File Manager `Create` procedure, or later using `SetFileInfo`).



A Document and its Stationery pad

- Be sure to use a different but similar icon for the stationery pad file. This is easy if you differentiate between stationery and normal files solely by file type—the Finder uses the type to determine which icon to display, see Technical Note #48 for help with the "bundle" mechanism used to associate a file type with an icon.

- When opening a stationery pad file, the window should come up named "Untitled", with the contents of the stationery pad file.

- "Revert" should re-read the stationery pad file.

- Don't forget to add the stationery pad's file type to the file-types list that you pass to Standard File, so that the new files will appear in the list when the user chooses Open. This file type should be registered with Macintosh Developer Technical Support.

#116: AppleShare-able Applications and the Resource Manager

See also: The Resource Manager
"Application Development in a Shared Environment"
Technical Note #40—Finder Flags

Written by: Bryan Stearns March 2, 1987
Updated: March 1, 1988

Normally, applications on an AppleShare server volume cannot be executed by more than one user at a time. This technical note explains why, and tells how you can enable your application to be shared.

The Resource Manager versus Shared Files

Part of the explanation of why applications are not automatically sharable is based on the design of the Resource Manager. The Resource Manager is a great little database. It was originally conceived as a way to keep applications localizable (a task it has performed admirably), and was found to be an excellent foundation for the Segment Loader, Font Manager, and a large part of the rest of the Macintosh operating system.

However, it was never designed to be a multi-user database. When the Resource Manager opens a resource file (such as an application), it reads the file's resource map into memory. This map remains in memory until the resource file is closed by the Segment Loader, which regains control when the application exits. Sometimes it is necessary to write the map out to disk; normally, this is only done by `UpdateResFile` and `CloseResFile`.

If two users opened the same resource file at the same time, and one of them had write access to the file and added a resource to it, the other user's Resource Manager wouldn't know about it; this would make the other user's copy of the file's original resource map invalid. This could cause (at least) a crash; if both users had write access, it's not unlikely that the resource file involved would become corrupted. Also, although you can tell the Resource Manager to write out an updated resource map, there's no way for another user to tell it to refresh the copy of the map in memory if the file changes.

What does all this have to do with running my application twice?

Your application is stored as a resource file; code segments, alert and dialog templates, etc., are resources. If you write to your application's resource file (for instance, to add configuration information, like print records), your application can't be shared.

In Apple's compatibility testing of existing applications (during development of AppleShare), we found quite a few applications, some of them quite popular, that wrote to their own resource files. So we decided, to improve the safety of using AppleShare, to always launch applications using a combination of access privileges such that only one user at a time could use a given application (these privileges will be discussed in a future Technical Note). In fact, AppleShare opens all resource files this way, unless the resource file is opened with `OpenRFPerm` and read-only permission is specified.

But my application doesn't write to itself!

We realize that many applications do not. However, there are other considerations (covered in detail, with suggestions for fixes, in "Application Development in a Shared Environment", available from APDA). In brief, here are the big ones we know about:

- Does your application create temporary files with fixed names in a fixed place (such as the directory containing the application)? Without AppleShare's protection, two applications trying to use the same temporary file could be disastrous.

- Is your application at least "conscious" of the fact that it may be in a multi-user environment? For instance, does it work correctly if a volume containing an existing document is on a locked volume? Does it check all result codes returned from File Manager calls, and `ResError` after relevant Resource Manager calls?

OK, I follow the rules. What do I do to make my application sharable?

There is a flag in each file's Finder information (stored in the file's directory entry) known as the "shared" bit. If you set this bit on your application's resource file, the Finder will launch your application using read-only permissions; if anyone else launches your application, they'll also get it read-only (their Finder will see the same "shared" bit set.).

Three important warnings accompany this information:

- The definition of the "shared" bit was incorrect in previous releases of information and software from Apple. This includes the June 16, 1986 version of Technical Note #40 (fixed in the March 2, 1987 version), as well as all versions of ResEdit before and including 1.1b3 (included with MPW 2.0). For now, the most reliable way to set this bit is to get the 1.1b3 version of ResEdit, use it to Get Info on your application, and check the box labeled "cached" (the incorrect documentation upon which ResEdit [et al.] was based called the real shared bit "cached"; the bit labeled as "shared" is the real cached bit [a currently unused but reserved bit which should be left clear]).

- By checking this bit, you're promising (to your users) that your application will work entirely correctly if launched by more than one user. This means that you follow the other rules, in addition to simply not writing to your application's own resource file. See "Application Development for a Shared Environment," and test carefully!

- Setting this bit has nothing to do with allowing your application's documents to be shared; you must design this feature into your application (it's not something that Apple system software can take care of behind your application's back.). You should realize from reading this note, however, that if you store your document's data in resource files, you won't be able to allow multiple users to access them simultaneously.

#117: Compatibility: Why & How

See Also: Technical Note #2—Compatibility Guidelines
Technical Note #7—A Few Quick Debugging Tips

Written by: Bo3b Johnson February 9, 1987
Updated: March 1, 1988

While creating or revising any program for the Macintosh, you should be aware of the most common reasons why programs fail on various versions of the Macintosh. This note will detail some common failure modes, why they occur, and how to avoid them.

We've tried to explain the issues in depth, but recognize that not everyone is interested in every issue. For example, if your application is not copy protected, you're probably not very interested in the section on copy protection. That's why we've included the outline form of the technical note. The first two pages outline the problems and the solutions that are detailed later. Feel free to skip around at will, but remember that we're sending this enormous technical note because the suggestions it provides may save you hasty compatibility revisions when we announce a new machine.

We know it's a lot, and we're here to help you if you need it. Our address (electronic and physical) is on page three—contact us with **any** questions—that's what we're here for!

Compatibility: the outline

Don't assume the screen is a fixed size
To get the screen size:
- check the QuickDraw global `screenBits.bounds`

Don't assume the screen is in a fixed location
To get the screen location:
- check the QuickDraw global `screenBits.baseAddr`

Don't assume that `rowBytes` is equal to the width of the screen
To get the number of bytes on a line:
- check the QuickDraw global `screenBits.rowBytes`

To get the screen width:
- check the QuickDraw global `screenBits.bounds.right`

To do screen-size calculations:
- Use `LongInts`

Don't write to or read from `nil` Handles or `nil` Pointers

Don't create or Use Fake Handles
To avoid creating or using fake handles:
- Always let the Memory Manager perform operations with handles
- Never write code that assigns something to a master pointer

Don't write code that modifies itself
Self modifying code will not live across incarnations of the 68000

Think carefully about code designed strictly as copy protection
To avoid copy protection-related incompatibilities:
- Avoid copy protection altogether
- Rely on schemes that don't require specific hardware
- Make sure your scheme doesn't perform illegal operations

Don't ignore errors
To get valuable information:
- Check all pertinent calls for errors
- Always write defensive code

Don't access hardware directly
To avoid hardware-related incompatibilities:
- Don't read or write the hardware
- If you can't get the support from the ROM, ask the system where the hardware is
- Use low-memory globals

Don't use bits that are reserved
To avoid compatibility problems when bit status changes:
- Don't use undocumented stuff
- When using low-memory globals, check only what you want to know

Summary

Minor bugs are getting harder and harder to get away with:

- Good luck
- We'll help
- AppleLink: MacDTS, MCI: MacDTS
- U.S. Mail: 20525 Mariani Ave.; M/S 27-T; Cupertino, CA 95014

What it Is

The basic idea is to make sure that your programs will run, regardless of which Macintosh they are being run on. The current systems to be concerned with include:

- Macintosh 128K
- Macintosh 512K
- Macintosh XL
- Macintosh 512Ke
- Macintosh Plus
- Macintosh SE
- Macintosh II

If you perform operations in a generic fashion, there is rarely any reason to know what machine is running. This means that you should avoid writing code to determine which version of the machine you are running on, unless it is absolutely necessary.

For the purposes of this discussion, the term "programs" will be used to describe any code that runs on a Macintosh. This includes applications, INITs, FKEYs, Desk Accessories and Drivers.

What the "Rules" mean

Compatibility across all Macintosh computers (which may sound like it involves more work for you) may actually mean that you have less work to do, since it may not be necessary to revise your program each time Apple brings out a new computer or System file. Users, as a group, do not understand compatibility problems; all they see is that the program does not run on their system.

The benefits of being compatible are many-fold: your customers/users stay happy, you have less programming to do, you can devote your time to more valuable goals, there are fewer versions to deal with, your code will probably be more efficient, your users will not curse you under their breath, and your outlook on life will be much merrier.

Now that we know what being compatible is all about, recognize that nobody is requiring you to be compatible with anything. Apple does not employ roving gangs of thought police to be sure that developers are following the recommended guidelines. Furthermore, when the guidelines comprise 1200 pages of turgid prose (*Inside Macintosh*), you can be expected to miss one or two of the "rules." It is no sin to be incompatible, nor is it a punishable offense. If it were, there would be no Macintosh programs, since virtually all developers would be incarcerated. What it does mean, however, is that your program will be unfavorably viewed until it steps in line with the current system (which is a moving target). If a program becomes incompatible with a new Macintosh, it usually requires rethinking the offending code, and releasing a new version. You may read something like "If the developers followed Apple guidelines, they would be compatible with the transverse-hinged diatomic quark realignment system." This means that if you made any mistakes (you read all 1200 pages carefully, right?), you will not be compatible. It is extremely difficult to remain completely compatible, particularly in a system as complex as the Macintosh. The rules haven't changed, but what you can get away with has. There are, however, a number of things that you can do to improve your odds—some of which will be explained here.

It's your choice

It is still your choice whether you will be concerned with compatibility or not. Apple will not put out a warrant for your arrest. However, if you are doing things that are specifically illegal, Apple will also not worry about "breaking" your program.

Bad Things

The following list is not intended to be comprehensive, but these are the primary reasons why programs break from one version of the system to the next. These are the current top ten commandments:

I Thou shalt not assume the screen is a fixed size.
II Thou shalt not assume the screen is at a fixed location.
III Thou shalt not assume that `rowBytes` is equal to the width of the screen.
IV Thou shalt not use `nil` handles or `nil` pointers.
V Thou shalt not create or use fake handles.
VI Thou shalt not write code that modifies itself.
VII Thou shalt think twice about code designed strictly as copy protection.
VIII Thou shalt check errors returned as function results.
IX Thou shalt not access hardware directly.
X Thou shalt not use any of the bits that are reserved (unused means reserved).

This has been determined from extensive testing of our diverse software base.

Assuming the screen is a fixed size

Do not assume that the Macintosh screen is 512 x 342 pixels. Programs that do generally have problems on (or special case for) the Macintosh XL, which has a wider screen. Most applications have to create the bounding rectangle where a window can be dragged. This is the `boundsRect` that is passed to the call:

```
DragWindow (myWindowPtr, theEvent.where, boundsRect);
```

Some ill-advised programs create the `boundsRect` by something like:

```
SetRect (boundsRect, 0,0,342,512);  { oops, this is hard-coded…}
```

Why it's Bad

This is bad because it is **never** necessary to specifically put in the bounding rectangle for the screen. On a Macintosh XL for example, the screen size is 760x364 (and sometimes 608x431 with alternate hardware). If a program uses the hard-coded 0,0,342,512 as a bounding rectangle, end users will not be able to move their windows past the fictitious boundary of 512. If something similar were done to the `GrowWindow` call, it would make it impossible for users to grow their window to fill the entire screen. (Always a saddening waste of valuable screen real-estate.)

Assuming screen size makes it more difficult to use the program on Macintoshes with big screens, by making it difficult to grow or move windows, or by drawing in strange places where they should not be drawing (outside of windows). Consider the case of running on a Macintosh equipped with one of the full page displays, or Ultra-Large screens. No one who paid for a big screen wants to be restricted to using only the upper-left corner of it.

How to avoid becoming a screening fascist

Never hard code the numbers 512 and 342 for screen dimensions. You should avoid using constants for system values that can change. Parameters like these are nearly always available in a dynamic fashion. Programs should read the appropriate variables while the program is running (at run-time, not at compile time).

Here's how smart programs get the screen dimensions:

```
InitGraf(@thePort); { QuickDraw global variables have to be initialized.}
…
boundsRect := screenBits.bounds;    { The Real way to get screen size }
                                    { Use QuickDraw global variable. }
```

This is smart, because the program never has to know specifically what the numbers are. All references to rectangles that need to be related to the screen (like the drag and grow areas of windows) should use `screenBits.bounds` to avoid worrying about the screen size.

Note that this does not do anything remotely like assume that "if the computer is not a standard Macintosh, then it must be an XL." Special casing for the various versions of the Macintosh has always been suspicious at best; it is now grounds for breaking. (At least with respect to screen dimensions.)

By the way, remember to take into account the menu bar height when using this rectangle. On 128K ROMs (and later) you can use the low-memory global `mBarHeight` (a word at `$BAA`). But since we didn't provide a low-memory global for the menu bar height in the 64K ROMs, you'll have to hard code it to 20 (`$14`). (You're not the only ones to forget the future holds changes.)

How to find fascist screenism in current programs

The easiest way is to exercise your program on one of the Ultra-Large screen Macintoshes. There should be no restrictions on sizing or moving the windows, and all drawing should have no problems. If there are any anomalies in the program's usage, there is probably a lurking problem. Also, do a global find in the source code to see if the numbers 512 or 342 occur in the program. If so, and if they are in reference to the screen, excise them.

Assuming the screen is at a fixed location

Some programs use a fixed screen address, assuming that the screen location will be the same on various incarnations of the Macintosh. This is not the case. For example, the screen is located at memory location `$1A700` on a 128K Macintosh, at `$7A700` on a 512K Macintosh, at `$F8000` on the Macintosh XL, and at `$FA700` on the Macintosh Plus.

Why it's Bad

When a program relies upon the screen being in a fixed location, Murphy's Law dictates that an unknowing user will run it upon a computer with the screen in a different location. This usually causes the system to crash, since the offending program will write to memory that was used for something important. Programs that crash have been proven to be less useful than those that don't.

How to avoid being a base screener

Suffice it to say that there is no way that the address of the screen will remain static, but there are rare occasions where it is necessary to go directly to the screen memory. On these occasions, there are bad ways and not-as-bad ways to do it. A bad way:

```
myScreenBase := Pointer ($7A700);  { not good.  Hard-coded number. }
```

A not-as-bad way:

```
InitGraf(@thePort);    { do this only once in a program. }
…
myScreenBase := screenBits.baseAddr;  { Good.  Always works. }
                                    {Yet another QuickDraw global variable}
```

Using the latter approach is guaranteed to work, since QuickDraw has to know where to draw, and the operating system tells QuickDraw where the screen can be found. When in doubt, ask QuickDraw. This will work on Macintosh computers from now until forever, so if you use this approach you won't have to revise your program just because the screen moved in memory.

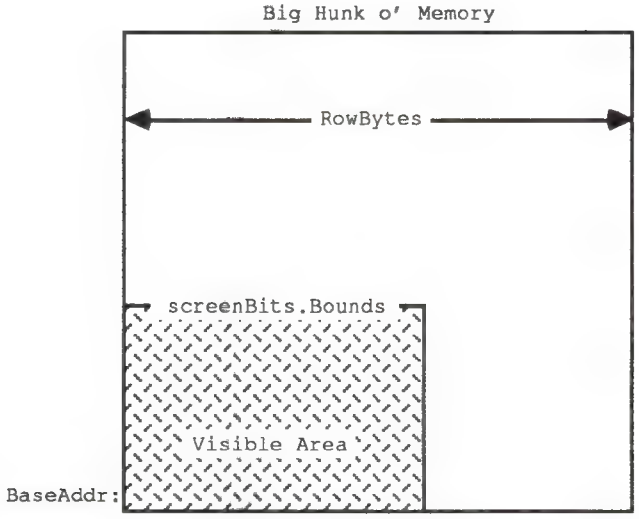
If you have a program (such as an INIT) that cannot rely upon QuickDraw being initialized (via `InitGraf`), then it is possible to use the `ScrnBase` low-memory global variable (a long word at `$824`). This method runs a distant second to asking QuickDraw, but is sometimes necessary.

How to find base screeners

The easiest way to find base screeners is to run the offending program on machines that have different screen addresses. If any addresses are being used in a base manner, the system will usually crash. The offending program may also occasionally refuse to draw. Some programs afflicted with this problem may also hang the computer (sometimes known as accessing funny space). Also, do a global find on the source code to look for numbers like `$7A700` or `$1A700`. When found, exercise caution while altering the offending lines.

Assuming that rowbytes is equal to the width of the screen

According to the definition of a `bitMap` found in *Inside Macintosh* (p I-144), you can see that `rowBytes` is the number of actual bytes in memory that are used to determine the `bitMap`. We know the screen is just a big hunk of memory, and we know that QuickDraw uses that memory as a `bitMap`. `rowBytes` accomplishes the translation of a big hunk of memory into a `bitMap`. To do this, `rowBytes` tells the system how long a given row is in memory and, more importantly, where in memory the next row starts. For conventional Macintoshes, `rowBytes` (bytes per Row) * 8 (Pixels per Byte) gives the final horizontal width of the screen as Pixels per Row. This does not have to be the case. It is possible to have a Macintosh screen where the `rowBytes` extends beyond what is actually visible on the screen. You can think of it as having the screen looking in on a larger `bitMap`. Diagrammatically, it might look like:



With an Ultra-Large screen, the number of bytes used for screen memory may be in the 500,000 byte range. Whenever calculations are being made to find various locations in the screen, the variables used should be able to handle larger screen sizes. For example, a 16 bit `Integer` will not be able to hold the 500,000 number, so a `LongInt` would be required. Do **not** assume that the screen size is 21,888 bytes long. `bitMaps` **can** be larger than 32K or 64K.

Why it's Bad

Programs that assume that all of the bytes in a row are visible may make bad calculations, causing drawing routines to produce unusual, and unreadable, results. Also, programs that use the `rowBytes` to figure out the width of the screen rectangle will find that their calculated rectangle is not the real `screenBits.Bounds`. Drawing into areas that are not visible will not necessarily crash the computer, but it will probably give erroneous results, and displays that don't match the normal output of the program.

Programs that assume that the number of bytes in the screen memory will be less than 32768 may have problems drawing into Ultra-Large screens, since those screens will often have more memory than a normal Macintosh screen. These particular problems do not evidence themselves by crashing the system. They generally appear as loss of

functionality (not being able to move a window to the bottom of the screen), or as drawing routines that no longer look correct. These problems can prevent an otherwise wonderful program from being used.

How to avoid being a row byter

In any calculations, the `rowBytes` variable should be thought of as the way to get to the next row on the screen. This is distinct from thinking of it as the width of the screen. The width should always be found from `screenBits.bounds.right-screenBits.bounds.left`.

It is also inappropriate to use the rectangle to decide how many bytes there are on a row. Programs that do something like:

```
bytesLine := screenBits.bounds.right DIV 8;  { bad use of bounds }
rightSide := screenBits.rowBytes * 8;     { bad use of rowBytes }
```

will find that the screen may have more `rowBytes` than previously thought. The best way to avoid being a row byter is to use the proper variables for the proper things. Without the proper mathematical basis to the screen, life becomes much more difficult. Always do things like:

```
bytesLine := screenBits.rowBytes;  { always the correct number }
rightSide := screenBits.bounds.right;  { always the correct screen size }
```

It is sometimes necessary to do calculations involving the screen. If so, be sure to use `LongInts` for all the math, and be sure to use the right variables (i.e. use `LongInts`). For example, if we need to find the address of the 500th row in the screen (500 lines from the top):

```
VAR   myAddress:   LongInt;
      myRow:       LongInt;      { so the calculations don't round off. }
      myOffset:    LongInt;      { could easily be over 32768 ... }
      bytesLine:   LongInt;

      ...
      myAddress := ord4(screenBits.baseAddr); {start w/the real base address }
      myRow := 500;                        {the row we want to address }
      bytesLine := screenBits.rowBytes;  {the real bytes per line }
      myOffset := myRow * bytesLine;      {lines * bytes per lines gives bytes }
      myAddress := myAddress + myOffset; {final address of the 500th line }
```

This is not something you want to do if you can possibly avoid it, but if you simply must go directly to the screen, be careful. The big-screen machines (Ultra-Large screens) will thank you for it. If QuickDraw cannot be initialized, there is also the low-memory global `screenRow` (a word at `$106`) that will give you the current `rowBytes`.

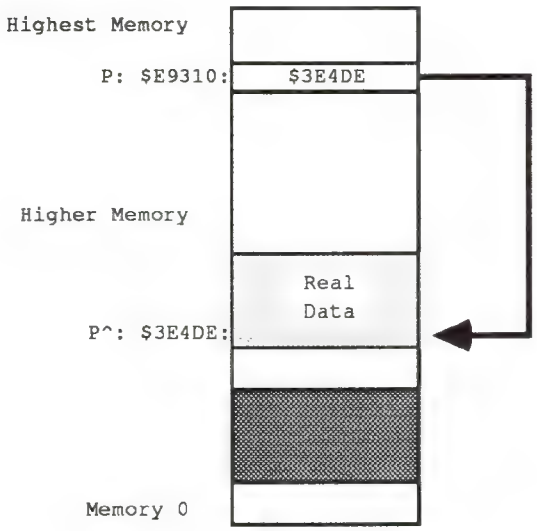
How to find row byters

To find current problems with row byter programs, run them on a machine equipped with Ultra-Large screens and see if any anomalies crop up. Look for drawing sequences that don't work right, and for drawing that clips to an imaginary edge. For source-level

inspection, look for uses of the `rowBytes` variables and be sure that they are being used in a mathematically sound fashion. Be highly suspicious of any code that uses `rowBytes` for the screen width. Any calculations involving those system variables should be closely inspected for round-off errors and improper use. Search for the number 8. If it is being used in a calculation where it is the number of bits per byte, then watch that code closely for improper conceptualization. This is code that could leap out and grab you by the throat at anytime. Be careful!

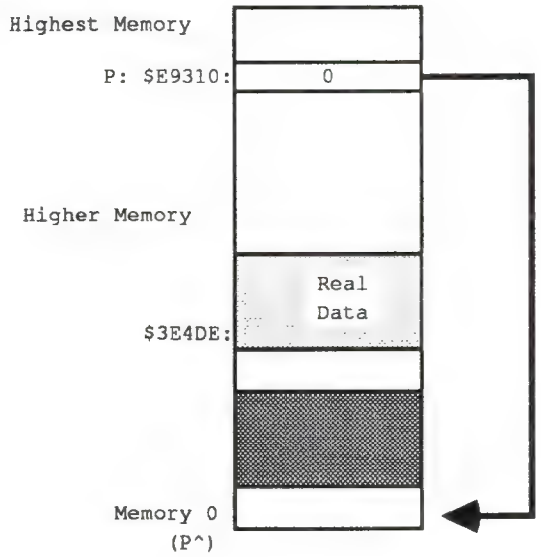
Using `nil` Handles or `nil` Pointers

A `nil` pointer is a pointer that has a value of 0. Recognize that pointers are merely addresses in memory. This means that a `nil` pointer is pointing to memory location 0. Any use of memory location 0 is strictly forbidden, since it is owned by Motorola. Trespassers may be shot on sight, but they may not die until much later. Sometimes trespassers are only wounded and act strangely. Any use of memory location 0 can be considered a bug, since there are **no** valid reasons for Macintosh programs to read or write to that memory. However, `nil` pointers themselves are not necessarily bad. It is occasionally necessary to pass `nil` pointers to ROM routines. This should not be confused with reading or writing to memory location 0. A pointer normally points to (contains the address of) a location in memory. It could look like this:



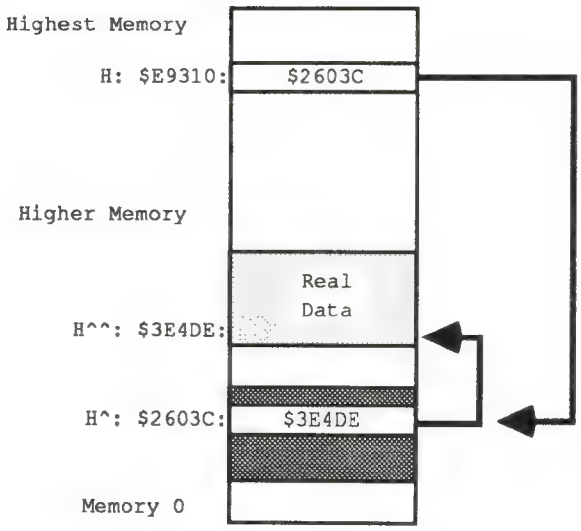
This is how a Pointer works. The address of the pointer variable itself is $E9310 (@P) and is four bytes long. The pointer points to (contains the address of) the block at $3E4DE (P). That memory location is where the actual data resides (P^).

If a pointer has been cleared to `nil`, it will point to memory location 0. This is OK as long as the program does not try to read from or write to that pointer. An example of a `nil` pointer could look like:



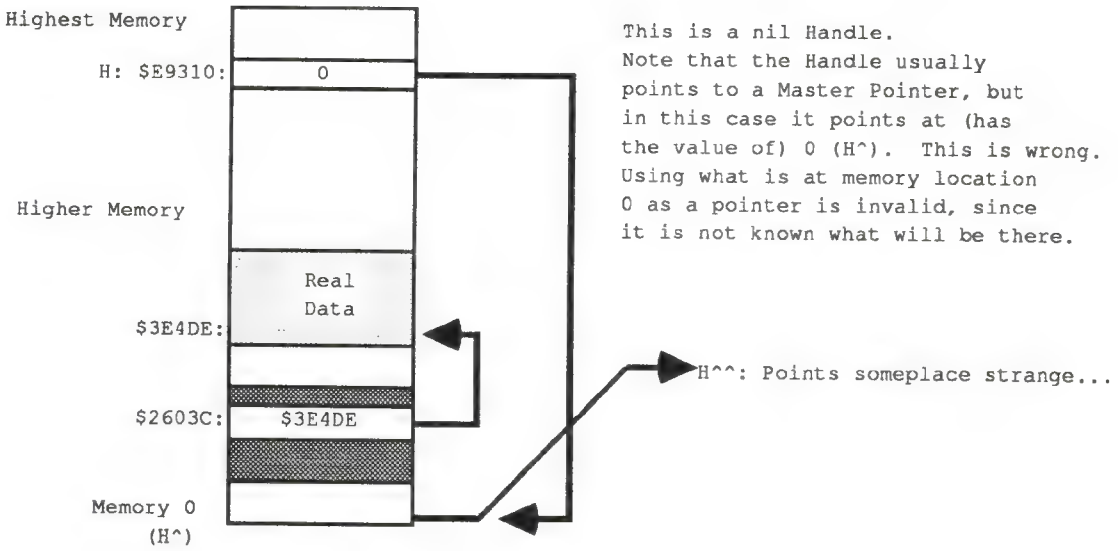
This is a nil Pointer. Note that the memory that it points to (the address) is 0 (P^). This is wrong. There is no valid data at memory location 0. Any writing to or reading from this pointer is a bug.

`nil` handles are related to the problem, since a handle is merely the address of a pointer (or a pointer to a pointer). An example of what a normal handle might look like is:



This is how a Handle works.
The address of the handle
variable itself (H) is $E9310.
That variable points (has the
address) to the master pointer
at location $2603C (H). That
variable is a pointer also, and
points to the real data found
at $3E4DE (H^^). The dark grey
block is a Master pointer block. It
is a group (usually 64) of Master
Pointers. One of them is the Master
Pointer at address $2603C (H^).

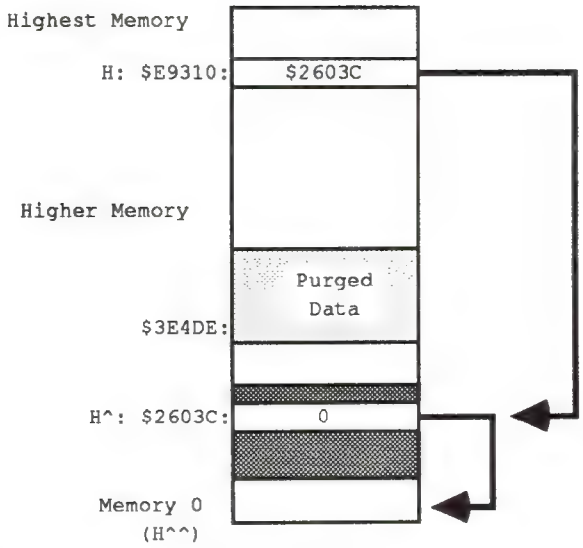
When the first pointer (`h`) becomes `nil`, that implies that memory location 0 can be used as a pointer. This is strictly illegal. There are **no** cases where it is valid to read from or write to a `nil` handle. A pictorial representation of what a `nil` handle could look like:



This is a nil Handle.
Note that the Handle usually
points to a Master Pointer, but
in this case it points at (has
the value of) 0 (H^). This is wrong.
Using what is at memory location
0 as a pointer is invalid, since
it is not known what will be there.

If the memory at 0 contains an odd number (numerically odd), then using it as a pointer will cause a system error with ID=2. This can be very useful, since that tells you exactly where the program is using this illegal handle, making it easy to fix. Unfortunately, there are cases where it is appropriate to pass a `nil` handle to ROM routines (such as `GetScrap`). These cases are rare, and it is **never** legal to read from or write to a `nil` handle.

There is also the case of an empty handle. An empty handle is one where the handle itself (the first pointer) points to a valid place in memory; that place in memory is also a pointer, and if it is `nil` the entire handle is termed empty. There are occasions where it is necessary to use the handle itself, but using the `nil` pointer that it contains is not valid. An example of an empty handle could be:



This is an Empty Handle. Note that the handle itself has a valid Master Pointer address in it $2603C (H^). The Master Pointer is nil however, which is the address of location 0 in memory. It is wrong to use the Master Pointer in this case, although there are cases where using the Handle itself is valid.

Fundamentally, any reading or writing to memory using a pointer or handle that is `nil` is punishable by death (of your program).

Why it's Bad

The use of `nil` pointers can lead to the use of make-believe data. This make-believe data often changes for different versions of the computer. This changing data makes it difficult to predict what will happen when a program uses `nil` pointers. Programs may not crash as a result of using a `nil` pointer, and they may behave in a consistent fashion. This does not mean that there isn't a bug. This merely means that the program is lucky, and that it should be playing the lottery, not running on a Macintosh. If a program acts differently on different versions of the Macintosh, you should think "could there be a nasty `nil` pointer problem here?" Use of a `nil` handle usually culminates in reading or writing to obscure places in memory. As an example:

```
VAR   myHandle:  TEHandle;

myHandle := nil;
```

That's pretty straightforward, so what's the problem? If you do something like:

```
myHandle^^.viewRect := myRect;  { very bad idea with myHandle = nil }
```

memory location zero will be used as a pointer to give the address of a TextEdit record. What if that memory location points to something in the system heap? What if it points to the sound buffer? In cases like these, eight bytes of rectangle data will be written to wherever memory location 0 points.

Use of a `nil` handle will never be useful. This memory is reserved and used by the 68000 for various interrupt vectors and Valuable Stuff. This Valuable Stuff is composed of things that you definitely do not want to change. When changed, the 68000 finds out, and decides to get back at your program in the most strange and wonderful ways. These strange results can range from a System Error all the way to erasing hard disks and destroying files. There really is no limit to the havoc that can be wreaked. This tends to keep the users on the edge of their seat, but this is not really the desired effect. As noted above, it won't necessarily cause traumatic results. A program can be doing naughty things and not get caught. This is still a bug that needs to be fixed, since it is nearly guaranteed to give different results on different versions of the Macintosh. Programs exhibiting schizophrenia have been proven to be less enjoyable to use.

How to avoid being a Niller

Whenever a program uses pointers and handles, it should ensure that the pointer or handle will not be `nil`. This could be termed defensive programming, since it assumes that everyone is out to get the program (which is not far from the truth on the Macintosh). You should always check the result of routines that claim to pass back a handle. If they pass you back a `nil` handle, you could get in trouble if you use them. Don't trust the ROM. The following example of a defensive use of a handle involves the Resource Manager. The Resource Manager passes back a handle to the resource data. There are any number of places where it may be forced to pass back a `nil` handle. For example:

```
VAR   myRezzie:  MyHandle;

myRezzie := MyHandle(GetResource(myResType, myResNumber)); { could be missing…}
IF myRezzie = nil  THEN  ErrorHandler('We almost got Nilled')
ELSE  myRezzie^^.myRect := newRect;              { We know it is OK }
```

As another example, think of how handles can be purged from memory in tight memory conditions. If a block is marked purgeable, the Memory Manager may throw it away at any time. This creates an empty handle. The defensive programmer will always make sure that the handles being used are not empty.

```
VAR   myRezzie:  myHandle;

myRezzie := myHandle(GetResource(myResType, myResNumber));  { could be
                                                              missing… }
IF myRezzie = nil  THEN  ErrorHandler('We almost got Nilled')
ELSE  myRezzie^^.myRect := newRect;     { We know it is OK }
tempHandle := NewHandle (largeBlock);  {might dispose a purgeable myRezzie}
IF myRezzie^ = nil  THEN LoadResource(Handle(myRezzie)); {Re-load empty
                                                          handle}
IF ResError = noErr  THEN
    myRezzie^^.StatusField := OK;       { guaranteed not empty, and actually
                                        gets read back in, if necessary }
```

Be especially careful of places where memory is being allocated. The `NewHandle` and `NewPtr` calls will return a `nil` handle or pointer if there is not enough memory. If you use that handle or pointer without checking, you will be guilty of being a Niller.

How to find Nillers

The best way to find these nasty `nil` pointer problems is to set memory location zero to be an **odd** number (a good choice is 'NIL!' = `$4E494C21`, which is numerically odd, as well as personality-wise). Please see Technical Note #7 for details on how to do this.

If you use TMON, you can use the extended user area with Discipline. Discipline will set memory location 0 to 'NIL!' to help catch those nasty pointer problems. If you use Macsbug, just type `SM 0 'NIL!` and go. Realize of course, that if a program has made a transgression and is actually using `nil` pointers, this may make the program crash with an ID=2 system error. This is good! This means that you have found a bug that may have been causing you untold grief. Once you know where a program crashes, it is usually very easy to use a debugger to find where the error is in the source code. When the program is compiled, turn on the debugging labels (usually a $D+ option). Set memory location 0 to be 'NIL!'. When the program crashes, look at where the program is executing and see what routine it was in (from a disassembly). Go back to that routine in the source code and remove the offending code with a grim smile on your face. Another scurvy bug has been vanquished. The intoxicating smell of victory wafts around your head.

Another way to find problems is to use a debugger to do a checksum on the first four bytes in memory (from 0 to 3 inclusive). If the program ever traps into the debugger claiming that the memory changed, see which part of the program altered memory location 0. Any code that writes to memory location zero is guilty of high treason against the state and must be removed. Remember to say, "bugs are not my friends."

Creating or Using Fake Handles

A fake handle is one that was not manufactured by the system, but was created by the program itself. An example of a fake handle is:

```
CONST aMem = $100;
VAR   myHandle: Handle;
      myPointer: Ptr;

myPointer := Ptr (aMem);      { the address of some memory }
myHandle := @myPointer;       {the address of the pointer variable. Very bad.}
```

The normal way to create and use handles is to call the Memory Manager `NewHandle` function.

Why it's Bad

A handle that is manufactured by the program is not a legitimate handle as far as the operating system is concerned. Passing a fake handle to routines that use handles is a good way to discover the meaning of "Death by ROM." For example, think how confused the operating system would get if the fake handle were passed to `DisposHandle`. What would it dispose? It never allocated the memory, so how can it release it? Programs that manufacture handles may find that the operating system is no longer their friend.

When handles are passed to various ROM routines, there is no telling what sorts of things will be done to the handle. There are any number of normal handle manipulation calls that the ROM may use, such as `SetHandleSize`, `HLock`, `HNoPurge`, `MoveHHi` and so on. Since a program cannot guarantee that the ROM will not be doing things like this to handles that the program passes in, it is wise to make sure that a real handle is being used, so that all these type of operations will work as the ROM expects. For fake handles, the calls like `HLock` and `SetHandleSize` have no bearing. Fake handles are very easy to create, and they are very bad for the health of otherwise upstanding programs. Whenever you need a handle, get one from the Memory Manager.

As a particularly bad use of a fake handle:

```
VAR   myHandle:  Handle;
      myStuff:  myRecord;

myHandle := NewHandle (SIZEOF(myStuff));    { create a new normal handle }
myHandle^ := @myStuff;  {YOW!  Intended to make myHandle a handle to
                           the myStuff record.  What it really does is
                           blow up a Master Pointer block, Heap corruption,
                           and death by Bad Heap.  Never do this. }
```

This can be a little confusing, since it is fine to use your own pointers, but very bad to use your own handles. The difference is that handles can move in memory, and pointers cannot, hence the pointers are not dangerous. This does not mean you should use pointers for everything since that causes other problems. It merely means that you have to be careful how you use the handles.

The use of fake handles usually causes system errors, but can be somewhat mysterious

in its effects. Fake handles can be particularly hard to track down since they often cause damage that is not uncovered for many minutes of use. Any use of fake handles that causes the heap to be altered will usually crash the system. Heap corruption is a common failure mode. In clinical studies, 9 out of 10 programmers recommend uncorrupted heaps to their users who use heaps.

How to avoid being a fakir

The correct way to make a handle to some data is to make a copy of the data:

```
VAR   myHandle:  Handle;
      myStuff:  myRecord;

errCode := PtrToHand (@myStuff, myHandle, SIZEOF(myStuff));
IF  errCode <> noErr  THEN ErrorHandler ('Out of memory');
```

Always, always, let the Memory Manager perform operations with handles. Never write code that assigns something to a master pointer, like:

```
VAR   myDeath:  Handle;
myDeath^ := stuff;  { Don't change the Master pointer. }
```

If there is code like this, it usually means the heap is being corrupted, or a fake handle is being used. It is, however, OK to pass around the handle itself, like:

```
myCopyHandle := myHandle;   { perfectly OK, nobody will yell about this. }
```

This is far different than using the ^ operator to accidentally modify things in the system. Whenever it is necessary to write code to use handles, be careful. Watch things carefully as they are being written. It is much easier to be careful on the way in than it is to try to find out why something is crashing. Be very careful of the @ operator. This operator can unleash untold problems upon unsuspecting programs. If at all possible, try to avoid using it, but if it is necessary, be absolutely sure you know what it is doing. It is particularly dangerous since it turns off the normal type checking that can help you find errors (in Pascal). In short, don't get crazy with pointer and handle manipulations, and they won't get crazy with you.

How to find fakirs

Problems of this form are particularly insidious because it can be very difficult to find them after they have been created. They tend to not crash immediately, but rather to crash sometime long after the real damage has been done. The best way to find these problems is to run the program with Discipline. (Discipline is a programmer's tool that will check all parameters passed to the ROM to see if they are legitimate. Discipline can be found as a stand-alone tool, but the most up-to-date version will be found in the Extended User Area for the TMON debugger. The User Area is public domain, but TMON itself is not. TMON has a number of other useful features, and is well worth the price.) Discipline will check handles that are passed to the ROM to see if they are real handles or not, and if not, will stop the program at the offending call. This can lead you back to the source at a point that may be close to where the bad handle was created. If a program passes the Discipline test, it will be a healthy, robust program with drastically

improved odds for compatibility. Programs that do not pass Discipline can sleep poorly at night, knowing that they have broken at least one or two of the "rules."

A way to find programs that are damaging the heap is to use a debugger (TMON or Macsbug) and turn on the Heap Check operation. This will check the heap for errors at each trap call, and if the heap is corrupted will break into the debugger. Hopefully this will be close to where the code is that caused the damage. Unfortunately, it may not be close enough; this will force you to look further back.

Looking in the source code, look for all uses of the `@` operator, and examine the code carefully to see if it is breaking the rules. If it is, change it to step in line with the rest of the happy programs here in happy valley. Also, look for any code that changes a master pointer like the `myHandle^ := stuff`. Any code of this form is highly suspect, and probably a member of the Anti-Productivity League. The APL has been accused of preventing software sales and the rise of the Yen. These problems can be quite difficult to find at times, but don't give up. These fake handles are high on the list of guilty parties, and should never be trusted.

Writing code that modifies itself

Self-modifying code is software that changes itself. Code that alters itself runs into two main groupings: code that modifies the code itself and code that changes the block the code is stored in. Copy protection code often modifies the code itself, to change the way it operates (concealing the meaning of what the code does). Changing the code itself is very tricky, and also prone to having problems, particularly when the microprocessor itself changes. There are third-party upgrades available that add a 68020 to a Macintosh. Because of the 68020's cache, programs that modify themselves stand a good chance of having problems when run on a 68020. This is a compatibility point that should not be missed (nudge, nudge, wink, wink). Code that changes other code (or itself) is prone to be incompatible when the microprocessor changes.

The second group is code that changes the block that the code is stored in. Keeping variables in the CODE segment itself is an example of this. This is uncommon with high-level languages, but it is easy to do in assembly language (using the DC directive). Variables defined in the code itself should be read-only (constants). Code that modifies itself has signed a tacit agreement that says "I'm being tricky, if I die, I'll revise it."

Why it's Bad

There are now three different versions of the microprocessor, the 68000, 68010, and the 68020. They are intended to be compatible with each other, but may not be compatible with code that modifies itself. As the Macintosh evolves, the system may have compatibility problems with programs that try to "push the envelope."

How to avoid being an abuser

Well, the obvious answer is to avoid writing self-modifying code. If you feel obliged to write self-modifying code, then you are taking an oath to not complain when you break in the future. But don't worry about accidentally taking the oath: you won't do it without knowing it. If you choose to abuse, you also agree to personal visits from the Apple thought police, who will be hired as soon as we find out.

How to find abusers

Run the program on a 68020 system. If it fails, it could be related to this problem, but since there are other bugs that might cause failures, it is not guaranteed to be a self-modifying code problem. Self-modifying code is often used in copy protection, which brings us to the next big topic.

Code designed strictly as copy protection

Copy protection is used to make it difficult to make copies of a program. The basic premise is to make it impossible to copy a program with the Finder. This will not be a discussion as to the pros and cons of copy protection. Everyone has an opinion. This will be a description of reality, as it relates to compatibility.

Why it's Bad

System changes will never be made merely to cause copy protection schemes to fail, but given the choice between improving the system and making a copy protection scheme remain compatible, the system improvement will always be chosen.

- Copy protection is number one on the list of why programs fail the compatibility test.
- Copy protection by its very nature tends to do the most "illegal" things.
- Programs that are copy protected are assumed to have signed a tacit agreement to revise the program when the system changes.

Copy protection itself is not necessarily bad. What is bad is when programs that would otherwise be fully compatible do not work due only to the copy protection. This is very sad, since it requires extra work, revisions to the software, and time lost while the revision is being produced. The users are not generally humored when they can no longer use their programs. Copy protection schemes that fail generally cause system errors when they are run. They also can refuse to run when they should.

How to avoid being a protectionist

The simple answer is to do without copy protection altogether. If you think of compatibility as a probability game, if you leave out the copy protection, your odds of winning skyrocket. As noted above, copy protection is the single biggest reason why programs fail on the various versions of the Macintosh. For those who are required to use copy protection, try to rely on schemes that do not require specific hardware and make sure that the scheme used is not performing illegal operations. If a program runs, an experienced Macintosh programmer armed with a debugger can probably make a copy of it, (no matter how sophisticated the copy protection scheme) so a moderate scheme that does not break the rules is probably a better compatibility bet. The trickier and more devious the scheme, the higher the chance of breaking a rule. Tread lightly.

How to find protectionists

The easiest way to see if a scheme is being overly tricky is to run it on a Macintosh XL. Since the floppy disk hardware is different this will usually demonstrate an unwanted hardware dependency. Be wary of schemes that don't allow installation on a hard disk. If the program cannot be installed on a hard disk, it may be relying upon things that are prone to change. Don't use schemes that access the hardware directly. All Macintosh software should go through the various managers in the ROM to maintain compatibility. Any code that sidesteps the ROM will be viewed as having said "It's OK to make me revise myself."

Check errors returned as function results

All of the Operating System functions, as well as some of the Toolbox functions, will return result codes as the value of the function. Don't ignore these result codes. If a program ignores the result codes, it is possible to have any number of bad things happen to the program. The result code is there to tell the program that something went wrong; if the program ignores the fact that something is wrong, that program will probably be killed by whatever went wrong. (Bugs do not like to be ignored.) If a program checks errors, an anomaly can be nipped in the bud, before something really bizarre happens.

Why it's Bad

A program that ignores result codes is skipping valuable information. This information can often prevent a program from crashing and keep it from losing data.

How to avoid becoming a skipper

Always write code that is defensive. Assume that everyone and everything is out to kill you. Trust no one. An example of error checking is:

```
myRezzie := GetResource (myResType, myResId);
IF  myRezzie = nil  THEN  ErrorHandler ('Who stole my resource...');
```

Another example:

```
fsErrCode := FSOpen ('MyFile', myVRefNum, myFileRefNum);
IF fsErrCode <> noErr  THEN ErrorHandler (fsErrCode, 'File error');
```

And another:

```
myTPPrPort := PrOpenDoc (myTHPrint, nil, nil);
IF  PRError <> noErr  THEN  ErrorHandler (PRError, 'Printing error');
```

Any use of Operating System functions should presume that something nasty can happen, and have code to handle the nasty situations. Printing calls, File Manager calls, Resource Manager calls, and Memory Manager calls are all examples of Operating System functions that should be watched for returning errors. Always, always check the result codes from Memory Manager calls. Big memory machines are pretty common now, and it is easy to get cavalier about memory, but realize that someone will always want to run the program under Switcher, or on smaller Macintoshes. It never hurts to check, and always hurts to ignore it.

How to find skippers

This is easy: just do weird things while the program is running. Put in locked or unformatted disks while the program is running. Use unconventional command sequences. Run out of disk space. Run on 128K Macintoshes to see how the program deals with running out of memory. Run under Switcher for the same reason. (Programs that die while running under Switcher are often not Switcher's fault, and are in fact due

to faulty memory management.) Print with no printer connected to the Macintosh. Pop disks out of the drives with the Command-Shift sequence, and see if the program can deal with no disk. When a disk-switch dialog comes up, press Command-period to pass back an error to the requesting program (128K ROMs only). Torturing otherwise well-behaved programs can be quite enjoyable, and a number of users enjoy torturing the program as much as the program enjoys torturing them. For the truly malicious, run the debugger and alter error codes as they come back from various routines. Sure it's a dirty low-down rotten thing to do to a program, but we want to see how far we can push the program. (This is also a good way to check your error handling.) It's one thing to be an optimist, but it's quite another to assume that nothing will go wrong while a program is running.

Accessing hardware directly

Sometimes it is necessary to go directly to the Macintosh hardware to accomplish a specific task for which there is no ROM support. Early hard disks that used the serial ports had no ROM support. Those disks needed to use the SCC chip (the 8530 communication chip) in a high-speed clocked fashion. Although it is a valid function, it is not something that is supported in the ROM. It was therefore necessary to go play with the SCC chip directly, setting and testing various hardware registers in the chip itself. Another example of a valid function that has no ROM support is the use of the alternate video page for page-flipping animation. Since there is no ROM call to flip pages, it is necessary to go play with the right bit in the VIA chip (6522 Versatile Interface Adapter). Going directly to the hardware does not automatically throw a program into the incompatible group, but it certainly lowers its odds.

Why it's bad

Going directly to the hardware poses any number of problems for enlightened programs that are trying to maintain compatibility across the various versions of the Macintosh. On the Macintosh XL for example, a lot of the hardware is found in different locations, and in some cases the hardware doesn't exist. On the XL there is no sound chip. Programs that go directly to the sound hardware will find they don't work correctly on an XL. If the same program were to go through the Sound Manager, it would work fine, although the sound would not be the same as expected. Since the Macintosh is heavily oriented to the software side of things, expecting various hardware to always be available is not a safe bet. Choosy programmers choose to leave the hardware to the ROM.

How to avoid having a hard attack

Don't read or write the hardware. Exhaust every possible conventional approach before deciding to really get down and dirty. If there is a Manager in the ROM for the operation you wish to perform, it is far better to use the Manager than to go directly to the hardware. Compatibility at the hardware level can very rarely be maintained, but compatibility at the Manager level is a prime consideration. If a program is down to the last ditch effort, and cannot get the support from the ROM that is desired, then access the hardware in an enlightened approach. The really bad way to do it:

```
VIA := Pointer ($EFE1FE);  { sure it's the base address today…}
                           { This is bad.  Hard-coded number. }
```

The with-it, inspired programmer of the eighties does something like:

```
TYPE LongPointer = ^LongInt;

VAR  VIA: LongPointer;
     VIABase: LongInt;

VIA := Pointer ($1D4);   { the address of the low-memory global. }
VIABase := VIA^;         { get the low-memory variable's value }
                        { Now VIABase has the address of the chip }
```

The point here is that the best way to get the address of a hardware chip is to ask the system where it currently is to be found. The system always knows where the pieces of the system are, and will always know for every incarnation of the Macintosh. There are low-memory global variables for all of the pieces of hardware currently found in the Macintosh. This includes the VIA, the SCC, the Sound Chip, the IWM, and the video display. Whenever you are stuck with going to the hardware, use the low-memory globals. The fact that a program goes directly to the hardware means that it is risking imminent incompatibility, but using the low-memory global will ensure that the program has the best odds. It's like going to Las Vegas: if you don't gamble at all, you don't lose any money; if you have to gamble, play the game that you lose the least on.

How to find hard attacks

Run the suspicious program on the Macintosh XL. Nearly all of the hardware is in a different memory location on the XL. If a program has a hard-coded hardware address in it, it will fail. It may crash, or it might not perform the desired task, but it won't work as advertised. This unfortunately, is not a completely legitimate test, since the XL does not have some of the hardware of other Macintoshes, and some of the hardware that is there has the register mapping different. This means that it is possible to play by the rule of using the low-memory global and still be incompatible.

Don't use bits that are reserved

Occasionally during the life of a Macintosh programmer, there comes a time when it is necessary to bite the bullet and use a low-memory global. These are very sad days, since it has been demonstrated (by history) that low-memory global variables are a mysterious lot, and not altogether friendly. One fellow in particular is known as ROM85, a word located at $28E. This particular variable has been documented as the way to determine if a program is running on the 128K ROMs or not. Notably, the top most bit of that word is the determining bit. This means that the rest of the bits in that word are reserved, since nothing is described about any further bits. Remember, if it doesn't say, assume it's reserved. If it's reserved, don't depend upon it. Take the cautious way out and assume that the other bits that aren't documented are used for Switcher local variables, or something equally wild. An example of a bad way to do the comparison is:

```
VAR  Rom85Ptr: WordPtr;
     RomsAre64: Boolean;

Rom85Ptr := Pointer ($28E);     { point at the low-memory global }
IF  Rom85Ptr^ = $7FFF  THEN  RomsAre64 := False  { Bad test. }
ELSE  RomsAre64 := True;
```

This is a bad test since the comparison is testing the value of **all** of the bits, not only the one that is valid. Since the other bits are undocumented, it is impossible to know what they are used for. Assume they are used for something that is arbitrarily random, and take the safe way out.

How to avoid being bitten

```
VAR      ROM85Ptr: Ptr

Rom85Ptr := Pointer ($28E);     { point at the low-memory global }
IF BitTst(ROM85Ptr,0) THEN RomsAre64 := True {Good--tests only hi-bit}
ELSE  RomsAre64 := False;
```

This technique will ensure that when those bits are documented, your program won't be using them for the wrong things. Beware of trojan bits.

Don't use undocumented stuff. Be very careful when you use anything out of the ordinary stream of a high-level language. For instance, in the ROM85 case, it is very easy to make the mistake of checking for an absolute value instead of testing the actual bit that encodes the information. Whenever a program is using low-memory globals, be sure that only the information desired is being used, and not some undocumented (and hence reserved) bits. It's not always easy to determine what is reserved and what isn't, so conservative programmers always use as little as possible. Be wary of the strange bits, and accept rides from none of them. The ride you take might cause you to revise your program.

How to find those bitten

Since there are such a multitude of possible places to get killed, there is no simple way to see what programs are using illegal bits. As time goes by it will be possible to find more of these cases by running on various versions of the Macintosh, but there will probably never be a comprehensive way of finding out who is accepting strange rides, and who is not. Whenever the use of a bit changes from reserved status to active, it will be possible to find those bugs via extensive testing. From a source level, it would be advisable to look over **any** use of low-memory globals, and eye them closely for inappropriate bit usage. Do a global search for the $ (which describes those ubiquitous hexadecimal numbers), and when found see if the use of the number is appropriate. Trust no one that is not known. If they are documented, they will stay where they are, and have the same meaning. Be very careful in realms that are undocumented. Bits that suddenly jump from reserved to active status have been known to cause more than one program to have a sudden anxiety attack. It is very unnerving to watch a program go from calm and reassuring to rabid status. Users have been known to drop their keyboards in sudden shock (which is bad on the keyboards).

Summary

So what does all this mean? It means that it is getting harder and harder to get away with minor bugs in programs. The minor bugs of yesterday are the major ones of today. No one will yell at you for having bugs in your program, since all programs have bugs of one form or another. The goal should be to make the programs run as smoothly and effortlessly as possible. The end-users will never object to bug-reduced programs.

What is the best way to test a program? A reasonably comprehensive test is to exercise all of the program's functions under the following situations:

- Use Discipline to be sure the program does not pass illegal things to the ROM.
- Use heap scramble and heap purge to be sure that handles are being used correctly, and that the memory management of the program is correct.
- Run with a checksum on memory locations 0...3 to see if the program writes to these locations.
- Run on a 128K Macintosh, or under Switcher with a small partition, to see how the program deals with memory-critical situations.
- Run on a 68020 system to see if the program is 68020-compatible and to make sure that changing system speed won't confuse the program.
- Run on a Macintosh XL to be sure that the program does not assume too much about the operating system, and to test screen handling.
- Run on an Ultra-Large screen to be sure that the screen handling is correct, and that there are no hard-coded screen dimensions.
- Run on 64K ROM machines to be sure new traps are not being used when they don't exist.
- Run under both HFS and MFS to be sure that the program deals with the file system correctly. (400K floppies are usually MFS.)

If a program can live through all of this with no Discipline traps, no checksum breaks, no system errors, no anomalies, no data loss and still get useful work done, then you deserve a gold medal for programming excellence. Maybe even an extra medal for conduct above and beyond the call of duty. In any case, you will know that you have done your job about as well as it can be done, with today's version of the rules, and today's programming tools.

Sounds like a foreboding task, doesn't it? The engineers in Macintosh Technical Support are available to help you with compatibility issues (we won't always be able to talk about new products, since we love our jobs, but we can give you some hints about compatibility with what the future holds).

Good luck.



#118: How To Check and Handle Printing Errors

Revised by: Pete "Luke" Alexander October 1990
Written by: Ginger Jernigan May 1987

This Technical Note formerly described how to check and properly handle errors that occur during printing with the Printing Manager.
Changes since March 1988: Merged contents into Technical Note #161.

This Note formerly described how to check and properly handle Printing Manager errors. This information is now contained in Technical Note #161, A Printing Loop That Cares..., which also includes a table of Printing Manager error codes

#119: Determining If Color QuickDraw Exists

See: Technical Note #129—SysEnvirons

Written by: Jim Friedlander May 4, 1987
Updated: March 1, 1988

This note formely described a way to determine if Color QuickDraw is present on a particular machine. We now recommend that you call `SysEnvirons` to find out, as described in Technical Note #129.

Macintosh
Technical Notes

Developer Technical Support

#120: Principia Off-Screen Graphics Environments

Updated by: Forrest Tanaka
Written by: Forrest Tanaka
Inspired by: Jim Friedlander, Rick Blair, and Rich Collyer

March 1992
October 1991

Using Color QuickDraw to draw off screen is a common requirement of applications and other kinds of programs that run on the Macintosh. This Note discusses what Color QuickDraw needs in a graphics environment and how to create one for off-screen drawing. A brief discussion of GWorlds, which are off-screen graphics environments that are set up by the system, is given to help you decide whether to use them or the do-it-yourself techniques described in this Note for setting up an off-screen graphics environment. The author's intent is to provide concepts and routines for creating an off-screen graphics environment, and also to explain why existing routines for off-screen drawing act as they do.

Many, many thanks go to Guillermo Ortiz, Konstantin Othmer, Bruce Leak, and Jon Zap for all their expertise on this subject, Rich Collyer, Rick Blair, and Jim Friedlander for paving the way, and especially to all people who inspired this update by asking great off-screen drawing questions.

Changes since October 1991: A very embarrassing bug was found in CreateOffScreen and UpdateOffScreen. If you try to create a 16- or 32-bit off-screen graphics environment, you'll just get a paramErr. It won't do that now.

Off-Screening

The Macintosh, as with every other CPU ever made by Apple, has memory-mapped video. That is, what you see on the screen is just the visual representation of a part of memory that's reserved for the video hardware (that's stretching the truth just a bit in the case of the text screens of the original Apple computer, the Apple II line, and the Apple III because there's also a character generator in those, but the overall process still looks roughly the same). If you change the contents of a memory location in this part of memory, then you'll see the corresponding location on the screen change when the video hardware draws the next frame or field of video. The resident raster graphics package, QuickDraw in the case of the Macintosh, draws images by stuffing the right values into the right places in the part of memory reserved for the video display. The resulting image on the screen looks like a line or perhaps an oval if you asked QuickDraw to draw a line or an oval, or it could be an entire complex image if you asked QuickDraw to draw one. This is normal, on-screen drawing.

Because video memory is a part of RAM just like any other part of RAM in the memory map of the Macintosh (or almost like; video memory might exist on a NuBus™ video card, but it's still RAM), QuickDraw can be told to draw into a part of memory that isn't reserved for the video hardware, maybe into a part of your own application's heap. When you tell QuickDraw to draw into a part of memory that's not reserved for the video hardware, you can't see any of the results. This is off-

screen drawing. There are plenty of perfectly good reasons to do this, such as providing storage for a paint-style document or to smoothly animate an image, but the assumption here is that you have a perfectly good reason to do this so you're more interested in the "how" of it instead of the "why" of it. If you need to know why, there are several books that cover off-screen drawing and the perfectly good reasons to do such a thing. A good place to start is Scott Knaster's book, *Macintosh Programming Secrets*, referenced at the end of this Note.

This Note is divided into these major sections:

- The introduction is the part that you're reading now.
- "The Building Blocks" provides an overview of the data structures that you need to tell Color QuickDraw to draw off screen.
- "Building the Blocks" discusses the construction and initialization of these data structures.
- "Playing With Blocks" shows an example of the use of these structures to draw off screen.
- "Put That Checkbook Away!" discusses some variations of these techniques to handle off-screen drawing for special cases.
- "The GWorld Factor" provides a brief overview of GWorlds, how to use them, and how they compare and contrast to the manual techniques that are described in most of this Note.

Those of you who aren't quite sure whether to use GWorlds or the do-it-yourself techniques might want to skip ahead for a moment to "The GWorld Factor" just in case doing it yourself is a waste of time. In any case, it's a good idea to read this whole Note because the concepts are mostly the same whether you're using GWorlds or not. GWorlds just make the process a lot easier, and they let you take advantage of the 8•24 GC video card. But, we're not in that section of the Note yet.

The Building Blocks

Before you can tell QuickDraw to draw off of the screen, you'll need to build three major data structures: a `CGrafPort`, a `PixMap`, and a `GDevice`. You'll also need a couple of tables that define the colors involved with drawing to and copying from the off-screen image: the color table and the inverse table. Of course, you'll need the pixel image itself, which is often called the "pixel buffer" or the "image buffer" or the "off-screen buffer" or just "the buffer." It's always called the "pixel image" in this Note. It doesn't necessarily buffer anything anyway.

The CGrafPort

A `CGrafPort` describes a drawing environment, and it's the color version of the `GrafPort` structure that's described on pages 147 through 155 in the QuickDraw chapter of *Inside Macintosh Volume* I. The drawing environment consists of, among other things, the size and location of the graphics pen, the foreground and background colors to use when something is drawn, the pattern to use, the region to clip all drawing to, and the portion of a pixel image that the `CGrafPort` logically exists in. Any initialized `CGrafPort` or `GrafPort` can be set as the current port through the `_SetPort` routine. The current port is a set of parameters that are implicitly passed to most QuickDraw routines.

The most important reason to build a new `CGrafPort` when you draw off screen rather than using an existing `CGrafPort` is so that switching between drawing to an off-screen graphics environment and drawing to one or more windows (each of which is an extended `GrafPort` or `CGrafPort` structure) on the screen is very easy. Some people use just one `CGrafPort` to share between on-screen and off-screen graphics environments, and switch their `PixMap` structures to switch between drawing on screen and drawing off screen. That does work, but if the off-screen and on-screen graphics environments have a different `clipRgn`, `visRgn`, pen characteristic, `portRect`, or any other characteristics that are different, then those must be switched at that time too. If you instead create a `CGrafPort` that's dedicated to one graphics environment, then a simple call to `_SetPort` effectively switches all these things for you at once. That's why every window on the screen comes with its own port. A simple call to `_SetPort` switches between the characteristics of each window even if each window has radically different drawing characteristics.

The `CGrafPort` data structure is more completely described in the "Color QuickDraw" chapter of *Inside Macintosh* Volume V, pages 49 through 52, and in the "Graphics Overview" chapter of *Inside Macintosh* Volume VI, pages 16-12 through 16-13.

The PixMap

A pixel image alone is just a formless blob of memory. Pixel maps, defined by the `PixMap` structure, describe pixel images, giving them a form and structure that's suitable for Color QuickDraw to draw into them and copy from them. The `PixMap` structure tells you the dimensions and location in memory of the pixel image, its coordinate system, and the depth and format of the pixels. Pixel maps that describe indexed-color pixel images additionally describe the colors that are represented by the values of the pixels in the pixel image. This is done through the color table, also known as the color look-up table or CLUT. Color tables are attached to pixel maps through their `pmTable` field. Direct-color pixel images have pixel values that describe their own colors, and so color tables aren't needed for those.

The `PixMap` structure is described in the "Color QuickDraw" chapter of *Inside Macintosh* Volume V, pages 52 through 55, and in the "Graphics Overview" chapter of *Inside Macintosh* Volume VI, pages 16-11 through 16-12. The concept of direct-color and indexed-color pixels is described in this same chapter on pages 16-16 through 16-18, and also in the "Color QuickDraw" chapter of the same volume on pages 17-4 through 17-10.

The GDevice

Graphics devices, defined by the `GDevice` structure, describe color environments. They're the most misunderstood data structure when it comes to off-screen graphics environments for three major reasons: first, they're not originally documented as being relevant to humans; second, they look as though they're only for screens; and third, it looks as though color tables describe color environments. We can dispose of these myths here: graphics devices are documented as being useful to humanity in this Note at least; they're critically important for both on-screen and off-screen drawing; and color tables describe the colors in pixel images, *not* color environments.

What's all this about color environments? In theory, there are virtually three hundred trillion colors available with Color QuickDraw through the 48-bit `RGBColor` record. In reality, there are never this many colors available, and in fact there might be only two. Color QuickDraw maps the theoretical color that you specify to the pixel value of the closest available color in the current color environment. This can be done with a color table, but that's not very efficient. Finding the closest available color to an `RGBColor` in a color table means searching the entire color table for that one closest color. If that's done just once, then performance isn't much of an issue, but if it's done many times, the performance hit could be significant. A very bad case of this is `_CopyBits`, where every pixel value in the source image is converted to an `RGBColor` by looking it up in the color

table of the source PixMap. If the color table of the destination PixMap had to be searched to find the closest available color for every pixel in the source PixMap, then the performance of even the most straightforward _CopyBits call could be a lot slower than it has to be.

To avoid this performance hit, the current GDevice provides an inverse table and a device type which are used to determine the available set of colors. Inverse tables are anticolor tables. Where color tables give you a color for a given pixel value, inverse tables give you a pixel value for a given color. Every conceivable color table has a corresponding conceivable inverse table, just as every positive real number has a corresponding negative real number, or every Mr. Spock has a corresponding Mr. Spock with a goatee. The device type specifies whether the color environment uses the indexed-color, fixed-color, or direct-color model. In the direct-color model, the inverse table is empty. Only the indexed-color and direct-color models are described in this Note.

When you specify a color in an indexed-color environment, Color QuickDraw takes the RGBColor specification and converts it into a value that can be used as an index into the inverse table of the current GDevice. To do this conversion, Color QuickDraw takes the top few significant bits of each color component and combines them into part of a 16-bit word, blue bits in the least significant bits, green bits right above it, and the red bits right above green bits. Any unused bits are in the most significant bits of the 16-bit word. The resulting 16-bit word is used as an index into the inverse table. The value in the inverse table at that index is the pixel value which best represents that color in the current color environment. The number of bits of each component that are used is determined by what's called the "resolution" of the inverse table. Almost always, the resolution of an inverse table is four bits, meaning the most significant four bits of each component are used to form the index into the inverse table. Figure 1 shows how an RGBColor record is converted to an index into an inverse table when the inverse-table resolution is four.

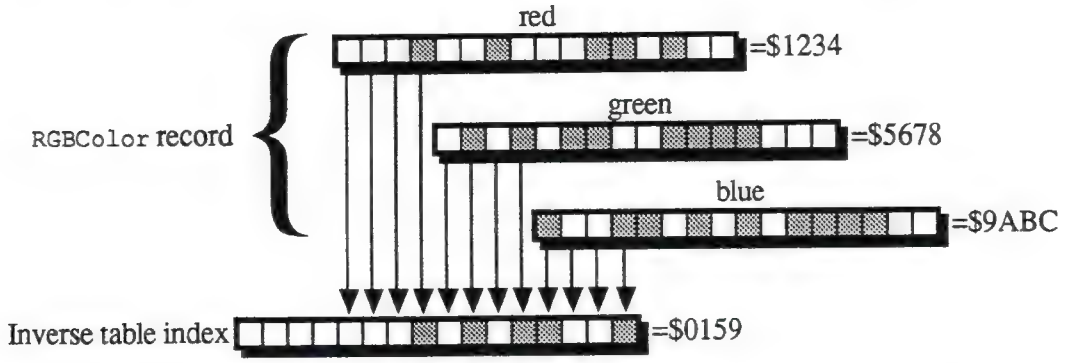


Figure 1 Conversion of RGBColor Record to Inverse-Table Index

The same process is used when _CopyBits is called with an indexed-color destination. Each pixel in the source pixel image is converted to an RGBColor either by doing a table look-up of the source pixel map's color table if the source pixel image uses indexed colors, or by expanding the pixel value to an RGBColor record if the source pixel image uses direct colors. The resulting RGBColor is then used to look up a pixel value in the inverse table of the current GDevice, and this pixel value is put into the destination pixel image.

If you specify a color in a direct-color environment, then the resulting RGBColor is converted to a direct pixel value by the processes that are shown on pages 17-6 through 17-9 of the "Color QuickDraw" chapter of *Inside Macintosh* Volume VI.

Usually, inverse-table look-up involves an extra step to find what are called "hidden colors" using proprietary information that's stored at the end of the inverse table. With an inverse-table resolution of four, only 16 shades of any particular component can be distinguished, and that's often not enough. An inverse table with a resolution of five is much larger, but it still only gives you 32 shades of any component. Hidden colors are looked up after the normal inverse-table look-up to give a much more accurate representation of the specified color in the current color environment than the inverse-table look-up alone can produce. Sometimes, most notably when the arithmetic transfer modes are used or if dithering is used, the hidden colors are ignored.

When a new color table is assigned to a `PixMap` or when its existing color table is modified, then a new corresponding inverse table should be generated for the `GDevice` that'll be used when drawing into that environment. Normally, this happens automatically without you having to do any more than inform Color QuickDraw of the change. This is described in more detail in "Changing the Off-Screen Color Table" later in this Note.

Graphics devices are documented in the "Graphics Devices" chapter of *Inside Macintosh* Volume VI which supersedes the "Graphics Devices" chapter of *Inside Macintosh* Volume V. They're also discussed in the "Graphics Overview" chapter of *Inside Macintosh* Volume VI, pages 16-13 through 16-14. The inverse-table mechanism is described in the "Color Manager" chapter of *Inside Macintosh* Volume V, pages 137 through 139.

All Together Now

There are a lot of different ways to put the three structures together, and this Note discusses the architecture that's shown in Figure 2. This architecture is useful when you want a simple, atomic, off-screen graphics environment.

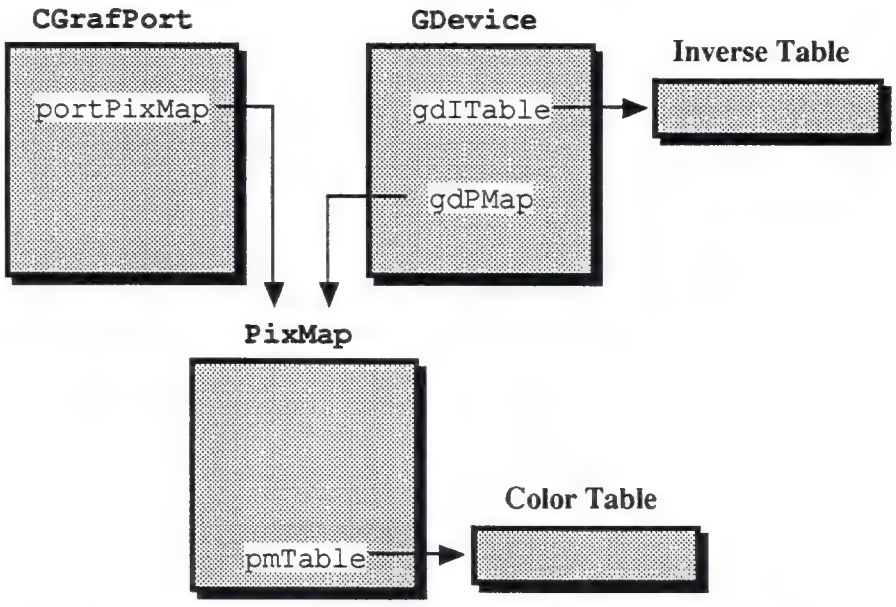


Figure 2 Relationships Between Structures for Off-Screen Drawing

Notice that there's no way to get to the `GDevice` from the `CGrafPort`, nor is there a way to get to the `CGrafPort` from the `GDevice`, though the `PixMap` can be found through either one. Your application must keep track of both the `CGrafPort` and the `GDevice`.

Building the Blocks

As with just about any algorithm, there are many ways to put the different structures together that form an off-screen graphics environment. This section covers just one way to build the architecture that's shown in Figure 2.

Building the CGrafPort

The `CGrafPort` structure is the easiest one to put together because the `_OpenCPort` routine initializes so many of the fields of the `CGrafPort` structure for you. It also allocates and initializes the structures that are attached to every `CGrafPort`, such as the `visRgn`, `clipRgn`, `grafVars` handle, and so forth. Most of these are initialized with values that are fine for general purposes, but the `visRgn`, `clipRgn`, and `portRect` fields should be set to the desired boundary rectangle of the off-screen graphics environment. What follows is an overview of each of the fields that you have to worry about when you're setting up a `CGrafPort` for drawing off screen.

`portPixMap` — handle to the off-screen `PixMap`. `_OpenCPort` initializes this field to a copy of the `PixMap` that's attached to the `gdPMap` field of the current `GDevice`. An overview of setting up this `PixMap` for drawing off screen is given in "Building the PixMap" later in this Note.

`portRect` — specifies the rectangular area of the associated pixel image that this `CGrafPort` controls. This field should be set to the desired rectangular area of the off-screen image because `_OpenCPort` doesn't necessarily initialize it to this size. Usually, the top-left corner of this rectangle has the coordinates (0, 0), but not necessarily so.

`visRgn` — handle to the region that specifies the visible area into which you can draw. `_OpenCPort` doesn't necessarily initialize it to the size of the off-screen image, so it should be set to the same size and coordinates as the `portRect` and left at that. This field is more important for windows because parts of them can be hidden by other windows.

`clipRgn` — handle to the region that specifies the logical area into which you can draw. `_OpenCPort` initializes it to cover the entire QuickDraw coordinate plane. It's usually a good idea to set it to the same size and coordinates as the `portRect` to avoid problems if the `clipRgn` is scaled or translated, which causes its signed integer coordinates to overflow and turn it into an empty region. One of the most common cases of this occurs when a picture that's created in this `CGrafPort` is drawn into a destination rectangle that's any larger than or translated from the original picture frame. Everything in the picture, including the clip region, is scaled to fit the destination rectangle. If the clip region covers the entire QuickDraw coordinate plane, then its coordinates overflow their signed integer bounds, and the clip region becomes logically empty. The result is that nothing is drawn.

The CreateOffScreen routine in Listing 1 creates an off-screen graphics environment, given a boundary rectangle, pixel depth, and color table, and it returns a new off-screen `CGrafPort` and `GDevice`, along with an error code. The desired pixel depth in bits per pixel is given in the `depth` parameter. If the pixel depth is eight or less, then an indexed-color graphics environment is created and a color table is required in the `colors` parameter. If the pixel depth is 16 or 32 bits per pixel and 32-Bit QuickDraw is available, then a direct-color graphics environment is created and the

colors parameter is ignored. If 32-Bit QuickDraw isn't available, then a pixel depth of 16 or 32 bits per pixel results in CreateOffScreen doing nothing more than returning a parameter error. A description of CreateOffScreen is given following the listing.

MPW Pascal Listing 1

```
FUNCTION CreateOffScreen(
   bounds:         Rect;       {Bounding rectangle of off-screen}
   depth:          Integer;    {Desired number of bits per pixel in off-screen}
   colors:         CTabHandle; {Color table to assign to off-screen}
   VAR retPort:    CGrafPtr;   {Returns a pointer to the new CGrafPort}
   VAR retGDevice: GDHandle    {Returns a handle to the new GDevice}
   ): OSErr;

   CONST
      kMaxRowBytes = $3FFE; {Maximum number of bytes in a row of pixels}

   VAR
      newPort:     CGrafPtr;     {Pointer to the new off-screen CGrafPort}
      newPixMap:   PixMapHandle; {Handle to the new off-screen PixMap}
      newDevice:   GDHandle;     {Handle to the new off-screen GDevice}
      qdVersion:   LongInt;      {Version of QuickDraw currently in use}
      savedPort:   GrafPtr;      {Pointer to GrafPort used for save/restore}
      savedState:  SignedByte;   {Saved state of color table handle}
      bytesPerRow: Integer;      {Number of bytes per row in the PixMap}
      error:       OSErr;        {Returns error code}

BEGIN
   (* Initialize a few things before we begin *)
   newPort := NIL;
   newPixMap := NIL;
   newDevice := NIL;
   error := noErr;

   (* Save the color table's current state and make sure it isn't purgeable *)
   IF colors <> NIL THEN
      BEGIN
         savedState := HGetState(Handle(colors));
         HNoPurge(Handle(colors));
      END;

   (* Calculate the number of bytes per row in the off-screen PixMap *)
   bytesPerRow := ((depth * (bounds.right - bounds.left) + 31) DIV 32) * 4;

   (* Get the current QuickDraw version *)
   error := Gestalt(gestaltQuickdrawVersion, qdVersion);
   error := noErr;

   (* Make sure depth is indexed or depth is direct and 32-Bit QD installed *)
   IF (depth = 1) OR (depth = 2) OR (depth = 4) OR (depth = 8) OR
         (((depth = 16) OR (depth = 32)) AND (qdVersion >= gestalt32BitQD)) THEN
      BEGIN
         (* Maximum number of bytes per row is 16,382; make sure within range *)
         IF bytesPerRow <= kMaxRowBytes THEN
            BEGIN
               (* Make sure a color table is provided if the depth is indexed *)
               IF depth <= 8 THEN
                  IF colors = NIL THEN
                     (* Indexed depth and clut is NIL; is parameter error *)
                     error := paramErr;
            END
         ELSE
            (* # of bytes per row is more than 16,382; is parameter error *)
```

```
            error := paramErr;
      END
   ELSE
      (* Pixel depth isn't valid; is parameter error *)
      error := paramErr;

   (* If sanity checks succeed, then allocate a new CGrafPort *)
   IF error = noErr THEN
      BEGIN
         newPort := CGrafPtr(NewPtr(SizeOf (CGrafPort)));
         IF newPort <> NIL THEN
            BEGIN
               (* Save the current port *)
               GetPort(savedPort);

               (* Initialize the new CGrafPort and make it the current port *)
               OpenCPort(newPort);

               (* Set portRect, visRgn, and clipRgn to the given bounds rect *)
               newPort^.portRect := bounds;
               RectRgn(newPort^.visRgn, bounds);
               ClipRect(bounds);

               (* Initialize the new PixMap for off-screen drawing *)
               error := SetUpPixMap(depth, bounds, colors, bytesPerRow,
                     newPort^.portPixMap);
               IF error = noErr THEN
                  BEGIN
                     (* Grab the initialized PixMap handle *)
                     newPixMap := newPort^.portPixMap;

                     (* Allocate and initialize a new GDevice *)
                     error := CreateGDevice(newPixMap, newDevice);
                  END;

               (* Restore the saved port *)
               SetPort(savedPort);
            END
         ELSE
            error := MemError;
      END;

   (* Restore the given state of the color table *)
   IF colors <> NIL THEN
      HSetState(Handle(colors), savedState);

   (* One Last Look Around The House Before We Go… *)
   IF error <> noErr THEN
      BEGIN
         (* Some error occurred; dispose of everything we allocated *)
         IF newPixMap <> NIL THEN
            BEGIN
               DisposCTable(newPixMap^^.pmTable);
               DisposPtr(newPixMap^^.baseAddr);
            END;
         IF newDevice <> NIL THEN
            BEGIN
               DisposHandle(Handle(newDevice^^.gdITable));
               DisposHandle(Handle(newDevice));
            END;
         IF newPort <> NIL THEN
            BEGIN
               CloseCPort(newPort);
```

```
            DisposPtr(Ptr(newPort));
          END;
      END
   ELSE
      BEGIN
         (* Everything's OK; return refs to off-screen CGrafPort and GDevice *)
         retPort := newPort;
         retGDevice := newDevice;
      END;
   CreateOffScreen := error;
END;
```

MPW C Listing 1

```
#define kMaxRowBytes 0x3FFE /* Maximum number of bytes in a row of pixels */

OSErr CreateOffScreen(
    Rect       *bounds,     /* Bounding rectangle of off-screen */
    short      depth,       /* Desired number of bits per pixel in off-screen */
    CTabHandle colors,      /* Color table to assign to off-screen */
    CGrafPtr   *retPort,    /* Returns a pointer to the new CGrafPort */
    GDHandle   *retGDevice) /* Returns a handle to the new GDevice */
{
    CGrafPtr     newPort;     /* Pointer to the new off-screen CGrafPort */
    PixMapHandle newPixMap;   /* Handle to the new off-screen PixMap */
    GDHandle     newDevice;   /* Handle to the new off-screen GDevice */
    long         qdVersion;   /* Version of QuickDraw currently in use */
    GrafPtr      savedPort;   /* Pointer to GrafPort used for save/restore */
    SignedByte   savedState;  /* Saved state of color table handle */
    short        bytesPerRow; /* Number of bytes per row in the PixMap */
    OSErr        error;       /* Returns error code */

    /* Initialize a few things before we begin */
    newPort = nil;
    newPixMap = nil;
    newDevice = nil;
    error = noErr;

    /* Save the color table's current state and make sure it isn't purgeable */
    if (colors != nil)
    {
        savedState = HGetState( (Handle)colors );
        HNoPurge( (Handle)colors );
    }

    /* Calculate the number of bytes per row in the off-screen PixMap */
    bytesPerRow = ((depth * (bounds->right - bounds->left) + 31) >> 5) << 2;

    /* Get the current QuickDraw version */
    (void)Gestalt( gestaltQuickdrawVersion, &qdVersion );

    /* Make sure depth is indexed or depth is direct and 32-Bit QD installed */
    if (depth == 1 || depth == 2 || depth == 4 || depth == 8 ||
            ((depth == 16 || depth == 32) && qdVersion >= gestalt32BitQD))
    {
        /* Maximum number of bytes per row is 16,382; make sure within range */
        if (bytesPerRow <= kMaxRowBytes)
        {
            /* Make sure a color table is provided if the depth is indexed */
            if (depth <= 8)
              if (colors == nil)
                  /* Indexed depth and clut is NIL; is parameter error */
                  error = paramErr;
        }
```

```
        else
            /* # of bytes per row is more than 16,382; is parameter error */
            error = paramErr;
    }
    else
        /* Pixel depth isn't valid; is parameter error */
        error = paramErr;

    /* If sanity checks succeed, then allocate a new CGrafPort */
    if (error == noErr)
    {
        newPort = (CGrafPtr)NewPtr( sizeof (CGrafPort) );
        if (newPort != nil)
        {
            /* Save the current port */
            GetPort( &savedPort );

            /* Initialize the new CGrafPort and make it the current port */
            OpenCPort( newPort );

            /* Set portRect, visRgn, and clipRgn to the given bounds rect */
            newPort->portRect = *bounds;
            RectRgn( newPort->visRgn, bounds );
            ClipRect( bounds );

            /* Initialize the new PixMap for off-screen drawing */
            error = SetUpPixMap( depth, bounds, colors, bytesPerRow,
                    newPort->portPixMap );
            if (error == noErr)
            {
                /* Grab the initialized PixMap handle */
                newPixMap = newPort->portPixMap;

                /* Allocate and initialize a new GDevice */
                error = CreateGDevice( newPixMap, &newDevice );
            }

            /* Restore the saved port */
            SetPort( savedPort );
        }
        else
            error = MemError();
    }

    /* Restore the given state of the color table */
    if (colors != nil)
        HSetState( (Handle)colors, savedState );

    /* One Last Look Around The House Before We Go… */
    if (error != noErr)
    {
        /* Some error occurred; dispose of everything we allocated */
        if (newPixMap != nil)
        {
            DisposCTable( (**newPixMap).pmTable );
            DisposPtr( (**newPixMap).baseAddr );
        }
        if (newDevice != nil)
        {
            DisposHandle( (Handle)(**newDevice).gdITable );
            DisposHandle( (Handle)newDevice );
        }
        if (newPort != nil)
```

```
        {
            CloseCPort( newPort );
            DisposPtr( (Ptr)newPort );
        }
    }
    else
    {
        /* Everything's OK; return refs to off-screen CGrafPort and GDevice */
        *retPort = newPort;
        *retGDevice = newDevice;
    }
    return error;
}
```

CreateOffScreen begins by making sure that the color table, if there is one, doesn't get purged during the time that the off-screen graphics environment is created. Then, a sanity check is done for the given depth, bounds, and color table. The depth must be either 1, 2, 4, or 8 bits per pixel, or additionally 16 or 32 bits per pixel if 32-Bit QuickDraw is available. If these conditions aren't satisfied, then it's decided that there's an error in the parameter list, and CreateOffScreen does nothing more. To determine whether 32-Bit QuickDraw is available or not, the `_Gestalt` routine is used. If `_Gestalt` returns a value that's equal to or greater than the constant `gestalt32BitQD`, then 32-Bit QuickDraw is available and depths of 16 and 32 bits per pixel are supported. It's not necessary to determine whether `_Gestalt` is available or not because it's implemented as glue code in the Macintosh Programmer's Workshop.

A check is then done to determine whether the number of bytes in each row of the off-screen pixel image is too much for QuickDraw to handle. Color QuickDraw can handle up to and including 16,382 ($3FFE) bytes in each row of any pixel image. If the required number of bytes per row exceeds this amount, then CreateOffScreen decides that there's an error in the parameter list and does nothing more. The minimum number of bytes in a row that's enough to cover the given boundary rectangle at the given pixel depth is calculated with the formula:

```
bytesPerRow := ((depth * (bounds.right - bounds.left) + 31) DIV 32) * 4;
```

This formula multiplies the number of pixels across the `PixMap` by the pixel depth to get the number of bits, and then this is divided by eight to get the number of bytes. This division by eight looks very strange because the number of bytes per row must be even, so this formula takes advantage of integer division and multiplication to make the result come out even. This particular formula additionally makes sure that the number of bytes per row is a multiple of four. This helps optimize the performance of Color QuickDraw operations because it allows Color QuickDraw to refer to each row beginning on a long word boundary in memory.

The last sanity check is to make sure that a color table is given as a parameter if it's needed. Indexed-color graphics environments need color tables, so if the given pixel depth is eight or less (which implies an indexed-color graphics environment) and the given color table is NIL, then CreateOffScreen decides that there's an error in the parameter list and does nothing more. If the given pixel depth is 16 or 32 (which implies a direct-color graphics environment), then CreateOffScreen ignores the given color table.

If all the sanity checks succeed, then the off-screen `CGrafPort` is allocated using a call to `_NewPtr`, and then it's initialized and opened as a `CGrafPort` by passing the resulting pointer to `_OpenCPort`. Because `_OpenCPort` makes the new `CGrafPort` the current port, the current port is first saved so that it can be restored as the current port when CreateOffScreen is done.

As mentioned above, the `_OpenCPort` doesn't necessarily initialize the `portRect`, `visRgn`, and `clipRgn` of the new `CGrafPort` to the areas that are needed for any particular off-screen graphics

environment. So, the given boundary rectangle is assigned to the `portRect` field, `_RectRgn` is called to make the `visRgn` equal to the given boundary rectangle, and `_ClipRect` is called to set the `clipRgn` so that it's equal to the given boundary rectangle.

The `PixMap` in the `portPixMap` field needs to be initialized for off-screen drawing, and that's handled by the SetUpPixMap routine that's described and defined in "Building the PixMap" later in this Note. Similarly, the off-screen `GDevice` must be created and initialized. That's handled by the CreateGDevice routine that's described and defined in "Building the GDevice" later in this Note.

Once these things are done, CreateOffScreen returns a pointer to the off-screen `CGrafPort` in the `retPort` parameter and a handle to the off-screen `GDevice` in the `retGDevice` parameter. The way to use these references is described in "Playing With Blocks" later in this Note.

Building the PixMap

`_OpenCPort` initializes the `portPixMap` field of the `CGrafPort` it's initializing with a copy of the `PixMap` of the current `GDevice`. When the CreateOffScreen routine described earlier executes, the current `GDevice` is unknown. So, all the fields of the `PixMap` that the new `CGrafPort` receives must be initialized so that it can be used for drawing off screen.* What follows is an overview of each of the `PixMap` fields and how they should be initialized for off-screen drawing.

`baseAddr` — pointer to the off-screen pixel image. The off-screen pixel image is allocated as a nonrelocatable block in the heap. The size of this block of memory is calculated from the `rowBytes` field, described next, multiplied by the number of rows in the given boundary rectangle.

`rowBytes` — number of bytes in each row of the pixel image. This value is calculated from the formula that's given in the CreateOffScreen routine. The most significant bit of this field should be set so that Color QuickDraw knows that this is a `PixMap` rather than a `BitMap`. The maximum value, ignoring the most significant bit, is 16,382.

`bounds` — defines the coordinate system and the dimensions of the pixel image. For most off-screen drawing, this should be a rectangle that covers the entire off-screen graphics environment.

`pmVersion` — set of internally and externally defined flags. As of 32-Bit QuickDraw 1.2, only the `baseAddr32` flag is defined externally. This flag is described in "Choosing Your Off-Screen Memory" later in this Note. For most off-screen drawing, this field is set to zero.

`packType` — image compression scheme for pictures. The options for this field are discussed in the "Graphics Overview" chapter of *Inside Macintosh* Volume VI, pages 17-22 through 17-23. In this Note, image compression isn't discussed so this field is set to zero.

---

* This part of these routines really bothers me because it feels impure to initialize all the `PixMap` fields when `_OpenCPort` has initialized them already, just not in a way that's any good for off-screen drawing. I tried creating the `GDevice` and `PixMap` first and then calling `_OpenCPort` so that it initializes its `PixMap` for off-screen drawing, but then you end up with two pixel maps and that makes this tougher to explain, or you have to dispose of one `PixMap` which seems worse than the method I'm using.

`packSize`	internally used field. This field is always set to zero.
`hRes`	horizontal resolution of the pixel map. By default, the QuickDraw resolution is 72 dots per inch,which is the value this Note uses. This is a fixed-point field, so the actual value in this field is $00480000.
`vRes`	vertical resolution of the pixel map. See the `hRes` description.
`pixelType`	format of the pixels. In indexed-color pixel maps, this field holds zero. In direct-color pixel maps, this field holds the `RGBDirect` constant, which is equal to 16.
`pixelSize`	number of bits in every pixel. For indexed-color pixels, this is 1, 2, 4, or 8 bits per pixel. For direct-color pixels, this is 16 or 32 bits per pixel.
`cmpCount`	number of components in every pixel. In indexed-color pixel maps, this field is set to 1. In direct-color pixel maps, this field is set to 3. Sometimes it's handy to set this field to 4 in 32-bit deep pixel maps when they're being saved in a picture. See the "Color QuickDraw" chapter of *Inside Macintosh* Volume VI, page 17-23, for details about this.
`cmpSize`	number of bits in each color component. In indexed-color pixel maps, this field is set to the same value that's in the `pixelSize` field. In 16-bit deep direct pixel maps, this field is set to 5. In 32-bit deep direct pixel maps, this field is set to 8.
`planeBytes`	not currently defined. This field is set to zero.
`pmTable`	handle to the color table for indexed-color pixel maps. A method to create a color table is given in "About That Creation Thing . . ." later in this Note. In direct-color pixel maps, this field contains a handle to a dummy color table, and building one of these is shown in the SetUpPixMap routine in Listing 2.
`pmReserved`	not currently defined. This field is set to zero.

The SetUpPixMap routine in Listing 2 initializes the `PixMap` that's passed to it in the `aPixMap` parameter so that it can be used in an off-screen graphics environment. The `depth`, `bounds`, and `color` parameters are the same as the ones passed to the CreateOffScreen routine. The `bytesPerRow` parameter is the number of bytes in each row of the off-screen pixel image. A description of SetUpPixMap follows the listing.

MPW Pascal Listing 2

```
FUNCTION SetUpPixMap(
   depth:       Integer;     {Desired number of bits/pixel in off-screen}
   bound:       Rect;        {Bounding rectangle of off-screen}
   colors:      CTabHandle;  {Color table to assign to off-screen}
   bytesPerRow: Integer;     {Number of bytes in each row of pixels}
   aPixMap:     PixMapHandle {Handle to the PixMap being initialized}
   ): OSErr;

   CONST
```

```
      kDefaultRes = $00480000; {Default resolution is 72 DPI; Fixed type}

   VAR
      newColors:   CTabHandle; {Color table used for the off-screen PixMap}
      offBaseAddr: Ptr;        {Pointer to the off-screen pixel image}
      error:       OSErr;      {Returns error code}

BEGIN
   error := noErr;
   newColors := NIL;
    offBaseAddr := NIL;

   (* Clone the clut if indexed color; allocate a dummy clut if direct color *)
   IF depth <= 8 THEN
      BEGIN
         newColors := colors;
         error := HandToHand(Handle(newColors));
      END
   ELSE
      BEGIN
         newColors := CTabHandle(NewHandle(SizeOf(ColorTable) -
               SizeOf(CSpecArray)));
         error := MemError;
      END;
   IF error = noErr THEN
      BEGIN
         (* Allocate pixel image; long integer multiplication avoids overflow *)
         offBaseAddr := NewPtr(LongInt(bytesPerRow) * (bound.bottom -
               bound.top));
         IF offBaseAddr <> NIL THEN
            WITH aPixMap^^ DO
               BEGIN
                  (* Initialize fields common to indexed and direct PixMaps *)
                  baseAddr := offBaseAddr;     {Point to image}
                  rowBytes := BOR(bytesPerRow, {MSB set for PixMap}
                        $8000);
                  bounds := bound;             {Use given bounds}
                  pmVersion := 0;              {No special stuff}
                  packType := 0;               {Default PICT pack}
                  packSize := 0;               {Always zero when in memory}
                  hRes := kDefaultRes;         {72 DPI default resolution}
                  vRes := kDefaultRes;         {72 DPI default resolution}
                  pixelSize := depth;          {Set number of bits/pixel}
                  planeBytes := 0;             {Not used}
                  pmReserved := 0;             {Not used}

                  (* Initialize fields specific to indexed and direct PixMaps *)
                  IF depth <= 8 THEN
                     BEGIN
                        (* PixMap is indexed *)
                        pixelType := 0;        {Indicates indexed}
                        cmpCount := 1;         {Have 1 component}
                        cmpSize := depth;      {Component size=depth}
                        pmTable := newColors; {Handle to CLUT}
                     END
                  ELSE
                     BEGIN
                        (* PixMap is direct *)
                        pixelType := RGBDirect; {Indicates direct}
                        cmpCount := 3;          {Have 3 components}
                        IF depth = 16 THEN
                           cmpSize := 5         {5 bits/component}
                        ELSE
```

```
                        cmpSize := 8;        {8 bits/component}

                    (* Initialize fields of the dummy color table *)
                    newColors^^.ctSeed := 3 * aPixMap^^.cmpSize;
                    newColors^^.ctFlags := 0;
                    newColors^^.ctSize := 0;
                    pmTable := newColors;
                END;
            END
        ELSE
            error := MemError;
    END
ELSE
    newColors := NIL;

(* If no errors occurred, return a handle to the new off-screen PixMap *)
IF error <> noErr THEN
    BEGIN
        IF newColors <> NIL THEN
            DisposCTable(newColors);
    END;

 (* Return the error code *)
 SetUpPixMap := error;
END;
```

MPW C Listing 2

```
#define kDefaultRes 0x00480000 /* Default resolution is 72 DPI; Fixed type */

OSErr SetUpPixMap(
    short        depth,       /* Desired number of bits/pixel in off-screen */
    Rect         *bounds,     /* Bounding rectangle of off-screen */
    CTabHandle   colors,      /* Color table to assign to off-screen */
    short        bytesPerRow, /* Number of bytes per row in the PixMap */
    PixMapHandle aPixMap)     /* Handle to the PixMap being initialized */
{
    CTabHandle newColors;   /* Color table used for the off-screen PixMap */
    Ptr        offBaseAddr; /* Pointer to the off-screen pixel image */
    OSErr      error;       /* Returns error code */

    error = noErr;
    newColors = nil;
    offBaseAddr = nil;

    /* Clone the clut if indexed color; allocate a dummy clut if direct color */
    if (depth <= 8)
    {
        newColors = colors;
        error = HandToHand( (Handle *)&newColors );
    }
    else
    {
        newColors = (CTabHandle)NewHandle( sizeof (ColorTable) -
                sizeof (CSpecArray) );
        error = MemError();
    }
    if (error == noErr)
    {
        /* Allocate pixel image; long integer multiplication avoids overflow */
        offBaseAddr = NewPtr( (unsigned long)bytesPerRow * (bounds->bottom -
                bounds->top) );
        if (offBaseAddr != nil)
        {
```

```
            /* Initialize fields common to indexed and direct PixMaps */
            (**aPixMap).baseAddr = offBaseAddr;  /* Point to image */
            (**aPixMap).rowBytes = bytesPerRow | /* MSB set for PixMap */
                    0x8000;
            (**aPixMap).bounds = *bounds;        /* Use given bounds */
            (**aPixMap).pmVersion = 0;           /* No special stuff */
            (**aPixMap).packType = 0;            /* Default PICT pack */
            (**aPixMap).packSize = 0;            /* Always zero in mem */
            (**aPixMap).hRes = kDefaultRes;      /* 72 DPI default res */
            (**aPixMap).vRes = kDefaultRes;      /* 72 DPI default res */
            (**aPixMap).pixelSize = depth;       /* Set # bits/pixel */
            (**aPixMap).planeBytes = 0;          /* Not used */
            (**aPixMap).pmReserved = 0;          /* Not used */

            /* Initialize fields specific to indexed and direct PixMaps */
            if (depth <= 8)
            {
                /* PixMap is indexed */
                (**aPixMap).pixelType = 0;       /* Indicates indexed */
                (**aPixMap).cmpCount = 1;        /* Have 1 component */
                (**aPixMap).cmpSize = depth;     /* Component size=depth */
                (**aPixMap).pmTable = newColors; /* Handle to CLUT */
            }
            else
            {
                /* PixMap is direct */
                (**aPixMap).pixelType = RGBDirect; /* Indicates direct */
                (**aPixMap).cmpCount = 3;          /* Have 3 components */
                if (depth == 16)
                    (**aPixMap).cmpSize = 5;       /* 5 bits/component */
                else
                    (**aPixMap).cmpSize = 8;       /* 8 bits/component */
                (**newColors).ctSeed = 3 * (**aPixMap).cmpSize;
                (**newColors).ctFlags = 0;
                (**newColors).ctSize = 0;
                (**aPixMap).pmTable = newColors;
            }
        }
        else
            error = MemError();
    }
    else
        newColors = nil;

    /* If no errors occurred, return a handle to the new off-screen PixMap */
    if (error != noErr)
    {
        if (newColors != nil)
            DisposCTable( newColors );
    }

    /* Return the error code */
    return error;
}
```

SetUpPixMap begins by copying the given color table if an indexed-color graphics environment is being built, or allocating a dummy color table if a direct-color graphics environment is being built. A copy of the color table is made because this allows the given color table and the off-screen graphics environment's color table to be manipulated independently without interfering with each other, and this lets the off-screen graphics environment routines manipulate the color table without needing to worry about whether the color table is a 'clut' resource or not. The dummy color table is made so that routines which assume that every PixMap has a color table won't do something

catastrophic if they find a NIL color table. The off-screen pixel image is then allocated as a nonrelocatable block in the application's heap.

Some of the fields of a `PixMap` have to be initialized differently depending upon whether the indexed-color model or the direct-color model is being used. So, the fields that are the same regardless of the color model that's being used are assigned first. Then the desired pixel depth is compared to 8. If the depth is less than or equal to 8, then the rest of the fields are initialized for the indexed-color model. Otherwise, the rest of the fields are initialized for the direct color model. In the case of the direct-color model, the dummy color table is initialized to have no `CSpecArray` entries and its `ctSeed` field is set to three times the component size. This dummy color table is then installed into the `PixMap`.

Once SetUpPixMap completes, the `PixMap` of the new `CGrafPort` is ready to hold an off-screen image. It's not quite ready to be drawn into with Color QuickDraw though. To do that, the off-screen `GDevice` is still needed; the construction and initialization of the `GDevice` are covered in the next section.

Building the GDevice

The `_OpenCPort` routine automatically allocates and initializes a `PixMap`, and the SetUpPixMap routine reinitializes that existing `PixMap`. `_OpenCPort` doesn't allocate nor initialize a `GDevice`, so one has to be created from scratch. Pages 21-20 through 21-21 of "The Graphics Devices Manager" chapter of *Inside Macintosh* Volume VI describe the `_NewGDevice` routine. This routine seems as though it's the ticket to getting a `GDevice` for off-screen drawing, but it always allocates the new `GDevice` in the system heap. That's not so good because if your program unexpectedly quits or if you just forget to dispose of the `GDevice` before you quit for real, the `GDevice` gets orphaned in the system heap. To prevent this from happening, `_NewGDevice` should be ignored and the off-screen `GDevice` should instead be allocated and initialized from scratch. What follows is a description of how each field of the `GDevice` structure should be initialized.

`gdRefNum` reference number of video driver. Off-screen graphics environments don't need to have video drivers because there's no video device associated with them, so this field is set to zero.

`gdID` used to identify specific `GDevice` structures from color-search procedures. This isn't necessary for off-screen drawing, so this is normally set to zero.

`gdType` type of `GDevice`. This field is set to the constant `clutType` (equal to zero) for an indexed-color environment and set to the constant `directType` (equal to 2) for a direct-color environment.

`gdITable` handle to the inverse table. Initially, this field is set to an arbitrarily small handle. Later, the `_MakeITable` routine is used to resize and initialize this handle to a real inverse table.

`gdResPref` inverse-table resolution. When `_MakeITable` is called by QuickDraw, the value of this field is used as the inverse-table resolution. Almost all inverse tables have a resolution of 4. There are some cases when a inverse-table resolution of 5 is useful, particularly when the arithmetic transfer modes are used with `_CopyBits`. See "The GDevice" earlier in this Note.

`gdSearchProc` pointer to the color-search procedure. If a color-search procedure is needed, this field can be set later by calling the `_AddSearch` routine (see the "Color

Manager" chapter of *Inside Macintosh* Volume V, pages 145 through 147). Usually, this field is just set to NIL and left at that.

`gdCompProc` — pointer to the color-complement procedure. If a color-complement procedure is needed, this field can be set later by calling the `_AddComp` routine (see the "Color Manager" chapter of *Inside Macintosh* Volume V, pages 145 through 147). Usually, this field is set to NIL and left at that.

`gdFlags` — flags indicating certain states of the `GDevice`. This field should initially be set to zeroes. After the `GDevice` has been built, these flags can be set with the `_SetDeviceAttrs` routine (see the "Graphics Devices Manager" chapter of *Inside Macintosh* Volume VI, pages 21-10 and 21-22).

`gdPMap` — handle to a `PixMap`. A handle to the `PixMap` of the `CGrafPort` that was created earlier is put into this field.

`gdRefCon` — miscellaneous data. `_CalcCMask` and `_SeedCFill` use this field as described on pages 71 through 72 of *Inside Macintosh* Volume V. Initially, this field is set to zero.

`gdNextGD` — handle to next `GDevice` in the `GDevice` list. The system maintains a linked list of `GDevice` records in which there's one `GDevice` for every screen, and the links are kept in this field. Off-screen `GDevice` structures should never be put into this list, so this field should be set to NIL.

`gdRect` — rectangle of `GDevice`. Strictly speaking, this field is used only for screens, but it should be the same as the `bounds` rectangle of the off-screen `PixMap`.

`gdMode` — current video mode. This field is used by video drivers to keep track of the current mode that the video device is in. For off-screen `GDevice` structures, this field should be set to -1.

`gdCC...` — These four fields are used only with `GDevice` structures for screens. For off-screen `GDevice` structures, these fields should be set to zero.

`gdReserved` — not currently defined. This field is set to zero.

The CreateGDevice routine shown below in Listing 3 allocates and initializes a `GDevice` structure. It takes the initialized off-screen `PixMap` in the `basePixMap` parameter and returns the initialized `GDevice` in the `retGDevice` parameter. If any error occurs, any memory that's allocated is disposed of and the result code is returned as a function result.

MPW Pascal Listing 3

```
FUNCTION CreateGDevice(
   basePixMap:     PixMapHandle; {Handle to the PixMap to base GDevice on}
   VAR retGDevice: GDHandle      {Returns a handle to the new GDevice}
   ): OSErr;

   CONST
      kITabRes = 4; {Inverse-table resolution}

   VAR
      newDevice:  GDHandle;   {Handle to the new GDevice}
      embryoITab: ITabHandle; {Handle to the embryonic inverse table}
```

```
      error:      OSErr;      {Error code}

BEGIN
   (* Initialize a few things before we begin *)
   error := noErr;
   newDevice := NIL;
   embryoITab := NIL;

   (* Allocate memory for the new GDevice *)
   newDevice := GDHandle(NewHandle(SizeOf(GDevice)));
   IF newDevice <> NIL THEN
      BEGIN
         (* Allocate the embryonic inverse table *)
         embryoITab := ITabHandle(NewHandleClear(2));
         IF embryoITab <> NIL THEN
            BEGIN
               (* Initialize the new GDevice fields *)
               WITH newDevice^^ DO
                  BEGIN
                     gdRefNum := 0;                {Only used for screens}
                     gdID := 0;                    {Won't normally use}
                     IF basePixMap^^.pixelSize <= 8 THEN
                        gdType := clutType         {Depth≤8; clut device}
                     ELSE
                        gdType := directType;      {Depth>8; direct device}
                     gdITable := embryoITab;       {2-byte handle for now}
                     gdResPref := kITabRes;        {Normal inv table res}
                     gdSearchProc := NIL;          {No color-search proc}
                     gdCompProc := NIL;            {No complement proc}
                     gdFlags := 0;                 {Will set these later}
                     gdPMap := basePixMap;         {Reference our PixMap}
                     gdRefCon := 0;                {Won't normally use}
                     gdNextGD := NIL;              {Not in GDevice list}
                     gdRect := basePixMap^^.bounds; {Use PixMap dimensions}
                     gdMode := -1;                 {For nonscreens}
                     gdCCBytes := 0;               {Only used for screens}
                     gdCCDepth := 0;               {Only used for screens}
                     gdCCXData := NIL;             {Only used for screens}
                     gdCCXMask := NIL;             {Only used for screens}
                     gdReserved := 0;              {Currently unused}
                  END;

               (* Set color-device bit if PixMap isn't black & white *)
               IF basePixMap^^.pixelSize > 1 THEN
                  SetDeviceAttribute(newDevice, gdDevType, true);

               (* Set bit to indicate that the GDevice has no video driver *)
               SetDeviceAttribute(newDevice, noDriver, true);

               (* Initialize the inverse table *)
               IF basePixMap^^.pixelSize <= 8 THEN
                  BEGIN
                     MakeITable(basePixMap^^.pmTable, newDevice^^.gdITable,
                           newDevice^^.gdResPref);
                     error := QDError;
                  END;
            END
         ELSE
            error := MemError;
      END
   ELSE
      error := MemError;

   (* Handle any errors along the way *)
   IF error <> noErr THEN
```

```
      BEGIN
         IF embryoITab <> NIL THEN
            DisposHandle(Handle(embryoITab));
         IF newDevice <> NIL THEN
            DisposHandle(Handle(newDevice));
      END
   ELSE
      retGDevice := newDevice;

   (* Return a handle to the new GDevice *)
   CreateGDevice := error;
END;
```

MPW C Listing 3

```
#define kITabRes 4 /* Inverse-table resolution */

OSErr CreateGDevice(
    PixMapHandle basePixMap,  /* Handle to the PixMap to base GDevice on */
    GDHandle     *retGDevice) /* Returns a handle to the new GDevice */
{
    GDHandle   newDevice;  /* Handle to the new GDevice */
    ITabHandle embryoITab; /* Handle to the embryonic inverse table */
    Rect       deviceRect; /* Rectangle of GDevice */
    OSErr      error;      /* Error code */

    /* Initialize a few things before we begin */
    error = noErr;
    newDevice = nil;
    embryoITab = nil;

    /* Allocate memory for the new GDevice */
    newDevice = (GDHandle)NewHandle( sizeof (GDevice) );
    if (newDevice != nil)
    {
        /* Allocate the embryonic inverse table */
        embryoITab = (ITabHandle)NewHandleClear( 2 );
        if (embryoITab != nil)
        {
            /* Set rectangle of device to PixMap bounds */
            deviceRect = (**basePixMap).bounds;

            /* Initialize the new GDevice fields */
            (**newDevice).gdRefNum = 0;            /* Only used for screens */
            (**newDevice).gdID = 0;                /* Won't normally use */
            if ((**basePixMap).pixelSize <= 8)
                (**newDevice).gdType = clutType;   /* Depth≤8; clut device */
            else
                (**newDevice).gdType = directType; /* Depth>8; direct device */
            (**newDevice).gdITable = embryoITab;   /* 2-byte handle for now */
            (**newDevice).gdResPref = kITabRes;    /* Normal inv table res */
            (**newDevice).gdSearchProc = nil;      /* No color-search proc */
            (**newDevice).gdCompProc = nil;        /* No complement proc */
            (**newDevice).gdFlags = 0;             /* Will set these later */
            (**newDevice).gdPMap = basePixMap;     /* Reference our PixMap */
            (**newDevice).gdRefCon = 0;            /* Won't normally use */
            (**newDevice).gdNextGD = nil;          /* Not in GDevice list */
            (**newDevice).gdRect = deviceRect;     /* Use PixMap dimensions */
            (**newDevice).gdMode = -1;             /* For nonscreens */
            (**newDevice).gdCCBytes = 0;           /* Only used for screens */
            (**newDevice).gdCCDepth = 0;           /* Only used for screens */
            (**newDevice).gdCCXData = 0;           /* Only used for screens */
            (**newDevice).gdCCXMask = 0;           /* Only used for screens */
```

```
            (**newDevice).gdReserved = 0;          /* Currently unused */

            /* Set color-device bit if PixMap isn't black & white */
            if ((**basePixMap).pixelSize > 1)
                SetDeviceAttribute( newDevice, gdDevType, true );

            /* Set bit to indicate that the GDevice has no video driver */
            SetDeviceAttribute( newDevice, noDriver, true );

            /* Initialize the inverse table */
            if ((**basePixMap).pixelSize <= 8)
            {
                MakeITable( (**basePixMap).pmTable, (**newDevice).gdITable,
                        (**newDevice).gdResPref );
                error = QDError();
            }
        }
        else
            error = MemError();
    }
    else
        error = MemError();

    /* Handle any errors along the way */
    if (error != noErr)
    {
        if (embryoITab != nil)
            DisposHandle( (Handle)embryoITab );
        if (newDevice != nil)
            DisposHandle( (Handle)newDevice );
    }
    else
        *retGDevice = newDevice;

    /* Return a handle to the new GDevice */
    return error;
}
```

CreateGDevice begins by allocating the `GDevice` structure and an embryonic form of the inverse table in the current heap. The inverse table is allocated as two zero bytes for now; it'll be resized and initialized to be a real inverse table later in this routine. Then, each of the `GDevice` fields are initialized as described earlier.

After all the fields have been initialized, the `gdFlags` field is set through `_SetDeviceAttribute`. If the desired pixel depth is greater than 1, then the `gdDevType` bit is set. This indicates that the `GDevice` is for a color graphics environment. This bit should be set even if a gray-scale color table is used for this off-screen graphics environment. The `noDriver` bit is set because this is an off-screen `GDevice` and so there's no associated video device driver.

Finally, the inverse table is resized and initialized by calling the `_MakeITable` routine. A handle to the two-byte embryonic inverse table that was created earlier in CreateGDevice is passed as a parameter, as is a handle to the off-screen color table and the preferred inverse-table resolution.

All Fall Down

Now that we have a way to create an off-screen graphics environment, there has to be a way to get rid of it too. The DisposeOffScreen routine shown in Listing 4 does this. The CreateOffScreen routine returns an off-screen graphics environment that's represented by a `CGrafPort` and `GDevice`. The DisposeOffScreen routine takes the off-screen `CGrafPort` and `GDevice` and

deallocates all the memory that's associated with them including the `CGrafPort` and its dependent structures, the `GDevice`, the `PixMap`, the color table, and the inverse table.

MPW Pascal Listing 4

```
PROCEDURE DisposeOffScreen(
   doomedPort:    CGrafPtr; {Pointer to the CGrafPort we're getting rid of}
   doomedGDevice: GDHandle  {Handle to the GDevice we're getting rid of}
   );

   VAR
      currPort:    CGrafPtr; {Pointer to the current port}
      currGDevice: GDHandle; {Handle to the current GDevice}

BEGIN
   (* Check to see whether the doomed CGrafPort is the current port *)
   GetPort(GrafPtr(currPort));
   IF currPort = doomedPort THEN
      BEGIN
         (* It is; set current port to Window Manager CGrafPort *)
         GetCWMgrPort(currPort);
         SetPort(GrafPtr(currPort));
      END;

   (* Check to see whether the doomed GDevice is the current GDevice *)
   currGDevice := GetGDevice;
   IF currGDevice = doomedGDevice THEN
      (* It is; set current GDevice to the main screen's GDevice *)
      SetGDevice(GetMainDevice);

   (* Throw everything away *)
   doomedGDevice^^.gdPMap := NIL;
   DisposGDevice(doomedGDevice);
   DisposPtr(doomedPort^.portPixMap^^.baseAddr);
   IF doomedPort^.portPixMap^^.pmTable <> NIL THEN
      DisposCTable(doomedPort^.portPixMap^^.pmTable);
   CloseCPort(doomedPort);
   DisposPtr(Ptr(doomedPort));
END;
```

MPW C Listing 4

```
void DisposeOffScreen(
    CGrafPtr doomedPort,    /* Pointer to the CGrafPort to be disposed of */
    GDHandle doomedGDevice) /* Handle to the GDevice to be disposed of */
{
    CGrafPtr currPort;    /* Pointer to the current port */
    GDHandle currGDevice; /* Handle to the current GDevice */

    /* Check to see whether the doomed CGrafPort is the current port */
    GetPort( (GrafPtr *)&currPort );
    if (currPort == doomedPort)
    {
        /* It is; set current port to Window Manager CGrafPort */
        GetCWMgrPort( &currPort );
        SetPort( (GrafPtr)currPort );
    }

    /* Check to see whether the doomed GDevice is the current GDevice */
    currGDevice = GetGDevice();
    if (currGDevice == doomedGDevice)
        /* It is; set current GDevice to the main screen's GDevice */
```

```
        SetGDevice( GetMainDevice() );

    /* Throw everything away */
    (**doomedGDevice).gdPMap = nil;
    DisposGDevice( doomedGDevice );
    DisposPtr( (**doomedPort->portPixMap).baseAddr );
    if ((**doomedPort->portPixMap).pmTable != nil)
        DisposCTable( (**doomedPort->portPixMap).pmTable );
    CloseCPort( doomedPort );
    DisposPtr( (Ptr)doomedPort );
}
```

One mildly tricky aspect of this is that we shouldn't dispose of the current graphics environment. To prevent this, the current port is retrieved by a call to `_GetPort`. If it returns a pointer to the same port that DisposeOffScreen is disposing, then the current port is set to the Window Manager's `CGrafPort`. That was an arbitrary choice, but it's the most neutral. Similarly, the current `GDevice` is retrieved by a call to `_GetGDevice`. If it returns a handle to the same `GDevice` that DisposeOffScreen is disposing, then the current port is set to the main screen's `GDevice`. Again, that's an arbitrary, neutral choice.

The inverse table, `GDevice`, pixel image, and color table are disposed of. Before disposing of the color table, a check is first made to see whether it's NIL. That's because it's reasonable, though not normal, for the `PixMap` not to have even a dummy color table if the direct-color model is being used. Then the `CGrafPort` is closed which deallocates all the pieces associated with the `CGrafPort`, including the `PixMap`. Once this is done, all the structures that were created by calling CreateOffScreen are deallocated.

Playing With Blocks

Now that these four routines with two entry points can create and dispose of off-screen graphics environments, how are they used? There are several phases to using an off-screen graphics environment: creating it, drawing into it, switching between it and other off-screen and on-screen graphics environments, copying images to and from it, and disposing of it. Listing 5 shows a routine called ExerciseOffScreen which is a very basic example of all of these phases.

MPW Pascal Listing 5

```
PROCEDURE ExerciseOffScreen;

   CONST
      kOffDepth  = 8;    {Number of bits per pixel in off-screen environment}
      rGrayClut  = 1600; {Resource ID of gray-scale clut}
      rColorClut = 1601; {Resource ID of full-color clut}

   VAR
      grayPort:    CGrafPtr;   {Graphics environment for gray off screen}
      grayDevice:  GDHandle;   {Color environment for gray off screen}
      colorPort:   CGrafPtr;   {Graphics environment for color off screen}
      colorDevice: GDHandle;   {Color environment for color off screen}
      savedPort:   GrafPtr;    {Pointer to the saved graphics environment}
      savedDevice: GDHandle;   {Handle to the saved color environment}
      offColors:   CTabHandle; {Colors for off-screen environments}
      offRect:     Rect;       {Rectangle of off-screen environments}
      circleRect:  Rect;       {Rectangles for circle-drawing}
      count:       Integer;    {Generic counter}
      aColor:      RGBColor;   {Color used for drawing off screen}
      error:       OSErr;      {Error return from off-screen creation}
```

```
BEGIN
   (* Set up the rectangle for the off-screen graphics environments *)
   SetRect(offRect, 0, 0, 256, 256);

   (* Get the color table for the gray off-screen graphics environment *)
   offColors := GetCTable(rGrayClut);

   (* Create the gray off-screen graphics environment *)
   error := CreateOffScreen(offRect, kOffDepth, offColors, grayPort,
         grayDevice);

   IF error = noErr THEN
      BEGIN
         (* Get the color table for the color off-screen graphics environment *)
         offColors := GetCTable(rColorClut);

         (* Create the color off-screen graphics environment *)
         error := CreateOffScreen(offRect, kOffDepth, offColors, colorPort,
               colorDevice);

         IF error = noErr THEN
            BEGIN
               (* Save the current graphics environment *)
               GetPort(savedPort);
               savedDevice := GetGDevice;

               (* Set the current graphics environment to the gray one *)
               SetPort(GrafPtr(grayPort));
               SetGDevice(grayDevice);

               (* Draw gray-scale ramp into the gray off-screen environment *)
               FOR count := 0 TO 255 DO
                  BEGIN
                     aColor.red := count * 257;
                     aColor.green := aColor.red;
                     aColor.blue := aColor.green;
                     RGBForeColor(aColor);
                     MoveTo(0, count);
                     LineTo(255, count);
                  END;

               (* Copy gray ramp into color off-screen colorized with green *)
               SetPort(GrafPtr(colorPort));
               SetGDevice(colorDevice);
               aColor.red := $0000; aColor.green := $FFFF; aColor.blue := $0000;
               RGBForeColor(aColor);
               CopyBits(GrafPtr(grayPort)^.portBits,
                     GrafPtr(colorPort)^.portBits,
                     grayPort^.portRect,
                     colorPort^.portRect,
                     srcCopy + ditherCopy, NIL);

               (* Draw red, green, and blue circles *)
               PenSize(8, 8);
               aColor.red := $FFFF; aColor.green := $0000; aColor.blue := $0000;
               RGBForeColor(aColor);
               circleRect := colorPort^.portRect;
               FrameOval(circleRect);
               aColor.red := $0000; aColor.green := $FFFF; aColor.blue := $0000;
               RGBForeColor(aColor);
               InsetRect(circleRect, 20, 20);
               FrameOval(circleRect);
               aColor.red := $0000; aColor.green := $0000; aColor.blue := $FFFF;
```

```
                RGBForeColor(aColor);
                InsetRect(circleRect, 20, 20);
                FrameOval(circleRect);

                (* Copy the color off-screen environment to the current port *)
                SetPort(savedPort);
                SetGDevice(savedDevice);
                CopyBits(GrafPtr(colorPort)^.portBits, savedPort^.portBits,
                      colorPort^.portRect, savedPort^.portRect,
                      srcCopy, NIL);

                (* Dispose of the off-screen graphics environments *)
                DisposeOffScreen(grayPort, grayDevice);
                DisposeOffScreen(colorPort, colorDevice);
            END;
        END;
END;
```

MPW C Listing 5

```
#define kOffDepth  8    /* Number of bits per pixel in off-screen environment */
#define rGrayClut  1600 /* Resource ID of gray-scale clut */
#define rColorClut 1601 /* Resource ID of full-color clut */

void ExerciseOffScreen()
{
    CGrafPtr   grayPort;    /* Graphics environment for gray off screen */
    GDHandle   grayDevice;  /* Color environment for gray off screen */
    CGrafPtr   colorPort;   /* Graphics environment for color off screen */
    GDHandle   colorDevice; /* Color environment for color off screen */
    GrafPtr    savedPort;   /* Pointer to the saved graphics environment */
    GDHandle   savedDevice; /* Handle to the saved color environment */
    CTabHandle offColors;   /* Colors for off-screen environments */
    Rect       offRect;     /* Rectangle of off-screen environments */
    Rect       circleRect;  /* Rectangles for circle-drawing */
    short      count;       /* Generic counter */
    RGBColor   aColor;      /* Color used for drawing off screen */
    OSErr      error;       /* Error return from off-screen creation */

    /* Set up the rectangle for the off-screen graphics environments */
    SetRect( &offRect, 0, 0, 256, 256 );

    /* Get the color table for the gray off-screen graphics environment */
    offColors = GetCTable( rGrayClut );

    /* Create the gray off-screen graphics environment */
    error = CreateOffScreen( &offRect, kOffDepth, offColors,
            &grayPort, &grayDevice );

    if (error == noErr)
    {
        /* Get the color table for the color off-screen graphics environment */
        offColors = GetCTable( rColorClut );

        /* Create the color off-screen graphics environment */
        error = CreateOffScreen( &offRect, kOffDepth, offColors,
                &colorPort, &colorDevice );

        if (error == noErr)
        {
            /* Save the current graphics environment */
            GetPort( &savedPort );
            savedDevice = GetGDevice();
```

```
            /* Set the current graphics environment to the gray one */
            SetPort( (GrafPtr)grayPort );
            SetGDevice( grayDevice );

            /* Draw gray-scale ramp into the gray off-screen environment */
            for (count = 0; count < 256; ++count)
            {
                aColor.red = aColor.green = aColor.blue = count * 257;
                RGBForeColor( &aColor );
                MoveTo( 0, count );
                LineTo( 255, count );
            }

            /* Copy gray ramp into color off-screen colorized with green */
            SetPort( (GrafPtr)colorPort );
            SetGDevice( colorDevice );
            aColor.red = 0x0000; aColor.green = 0xFFFF; aColor.blue = 0x0000;
            RGBForeColor( &aColor );
            CopyBits( &((GrafPtr)grayPort)->portBits,
                    &((GrafPtr)colorPort)->portBits,
                    &grayPort->portRect,
                    &colorPort->portRect,
                    srcCopy | ditherCopy, nil );

            /* Draw red, green, and blue circles */
            PenSize( 8, 8 );
            aColor.red = 0xFFFF; aColor.green = 0x0000; aColor.blue = 0x0000;
            RGBForeColor( &aColor );
            circleRect = colorPort->portRect;
            FrameOval( &circleRect );
            aColor.red = 0x0000; aColor.green = 0xFFFF; aColor.blue = 0x0000;
            RGBForeColor( &aColor );
            InsetRect( &circleRect, 20, 20 );
            FrameOval( &circleRect );
            aColor.red = 0x0000; aColor.green = 0x0000; aColor.blue = 0xFFFF;
            RGBForeColor( &aColor );
            InsetRect( &circleRect, 20, 20 );
            FrameOval( &circleRect );

            /* Copy the color off-screen environment to the current port */
            SetPort( savedPort );
            SetGDevice( savedDevice );
            CopyBits( &((GrafPtr)colorPort)->portBits, &savedPort->portBits,
                    &colorPort->portRect, &savedPort->portRect,
                    srcCopy, nil );

            /* Dispose of the off-screen graphics environments */
            DisposeOffScreen( grayPort, grayDevice );
            DisposeOffScreen( colorPort, colorDevice );
        }
    }
}
```

Two off-screen graphics environments are created in the same way. A rectangle that's 256 pixels wide by 256 pixels high and with its top-left coordinate at (0, 0) is created in the `offRect` local variable. 'clut' resources are loaded from the application's resource fork to use as the color tables of the two off-screen graphics environments; a gray-scale 'clut' in the first case and a full-color 'clut' in the second case. Then, CreateOffScreen is called with the rectangle, color table, and a hard-coded pixel depth of eight bits per pixel.

If CreateOffScreen returns `noErr` in both cases, then the current graphics environment is saved so that it can be restored later. Graphics environments consist of the current port and the current `GDevice`. The current `GrafPort` or `CGrafPort` is saved with `_GetPort`. The current `GDevice` is saved with `_GetGDevice`.

The gray-scale off-screen graphics environment is set as the current graphics environment by calling `_SetPort` with its `CGrafPort` and calling `_SetGDevice` with its `GDevice`. A vertical gray ramp is drawn into this graphics environment with the usual set of QuickDraw calls. This graphics environment's pixel image is then copied to the full-color off-screen graphics environment with dithering and colorization with green (dithering requires 32-Bit QuickDraw and consistent colorization requires system software version 7.0; both of these features are described in Konstantin Othmer's article "QuickDraw's CopyBits Procedure: Better Than Ever in System 7.0" in Issue 6 of *develop*). Before this copy happens, the full-color off-screen graphics environment must be set as the current one. Once this is done, `_CopyBits` can properly map colors from the gray-scale off-screen graphics environment to the full-color one which gets a green ramp image.

Red, green, and blue concentric circles are drawn into the full-color off-screen graphics environment over the green ramp. This image is then copied to the graphics environment that was the current one when ExerciseOffScreen was called. To do this, the saved graphics environment is set as the current one by what should now be the familiar calls to `_SetPort` and `_SetGDevice`. The off-screen image is then copied to the saved graphics environment with `_CopyBits`.

Finally, the two off-screen graphics environments are disposed of by calling the DisposeOffScreen routine that's defined in the section "All Fall Down" earlier in this Note.

Put That Checkbook Away!

The previous section covered the basics of creating and using off-screen graphics environments. This is good enough for many, if not most, needs of off-screen drawing. But there are variations to creating and maintaining an off-screen graphics environment for specific cases. This section discusses a few of the more common cases.

About That Creation Thing . . .

The CreateOffScreen routine, defined in Listing 1, takes three pieces of information: the boundary rectangle, the desired pixel depth, and the desired color table. But there's much more to these pieces than ExerciseOffScreen shows. This section describes these pieces in more detail.

The first parameter to CreateOffScreen is a rectangle which determines the size and coordinate system of the off-screen graphics environment. Usually, the top-left corner of the rectangle has the coordinate (0, 0) because it's usually easiest to draw everything using coordinates that can also be thought of as the horizontal and vertical distance in pixels from the top-left corner of the graphics environment. But in some cases, it's more convenient to have the (0, 0) coordinate somewhere else, and passing CreateOffScreen a rectangle with a nonzero coordinate in the top-left corner is an easy way to do this. The coordinate system can be translated after the off-screen graphics environment is created by using the `_SetOrigin` routine that's described on pages 153 through 155 of *Inside Macintosh* Volume I.

Warning: As *Inside Macintosh* Volume I, page 154, notes, the clip region of the port "sticks" to the coordinate system when you call `_SetOrigin`. If `_SetOrigin` offsets the coordinate system by a large amount, then the clip region might be moved completely outside of the port's drawing area, and

nothing can be drawn into that port. After calling `_SetOrigin`, you should set the clip region so that you can continue drawing into the port.

The number of bits per pixel implies the maximum number of available colors in a graphics environment, at least roughly speaking. The relationship between the number of bits per pixel and the number of available colors is discussed in the "Graphics Overview" chapter of *Inside Macintosh* Volume VI, pages 16-8 through 16-9.

If an indexed-color graphics environment is being made, then a color table must be passed to CreateOffScreen. In ExerciseOffScreen, the color table is retrieved from a 'clut' resource that's in the application's resource fork with a call to `_GetCTable`. Because CreateOffScreen clones this color table, this 'clut' resource can be purgeable so that it can be thrown out if its memory is needed for other purposes. `_GetCTable` can also be passed some special constants that tell it to allocate various system color tables that can also be passed to CreateOffScreen. These special constants are described on page 17-18 of the "Color QuickDraw" chapter of *Inside Macintosh* Volume VI. `_GetCTable` allocates memory for these system color tables, so they should be disposed of after you're done with them.

A color table could also be built from scratch by allocating it with a call to `_NewHandle` and then initializing it by hand. The `ColorTable` structure is documented on pages 48 through 49 of *Inside Macintosh* Volume V. Here's what each of the fields should be set to:

`ctSeed` identification value. This is an arbitrary value that should be changed any time the contents of the color table change so that the inverse table can be kept current. When Color QuickDraw draws anything, it compares the `ctSeed` of the color table of the `PixMap` of the current `GDevice` against the `iTabSeed` field of the inverse table of the current `GDevice`. If they're the same, then Color QuickDraw uses colors according to that inverse table. If they're different, then Color QuickDraw first rebuilds the inverse table according to the new color table's contents and its `iTabSeed` is set to the value of the new color table's `ctSeed`; then the rebuilt inverse table is used.

When `_CopyBits` is called with the `srcCopy` transfer mode, the `ctSeed` fields of the source and destination pixel maps are compared. If they're the same, then `_CopyBits` simply transfers the source pixels to the destination with no mapping of colors. If they're different, then `_CopyBits` checks each entry of the color tables to determine whether they have the same colors for the same pixel values. If they do, then `_CopyBits` again simply transfers the source pixels to the destination with no mapping of colors. If they don't, then `_CopyBits` maps colors in the source `PixMap` to the colors in the current graphics environment according to the inverse table of the current `GDevice`. The `ctSeed` field of a color table should be changed whenever its contents are changed so that `_CopyBits` doesn't make the wrong assumptions about the equality of the source and destination color tables.

You can get a seed value for a new color table by assigning to it the result of the `_GetCTSeed` routine, documented in the "Color Manager" chapter of *Inside Macintosh* Volume V, page 143. If the contents of an existing color table are changed, then it should be passed to the `_CTabChanged` routine which assigns a new value to its `ctSeed` field. If the `_CTabChanged` routine isn't available (it's available with 32-Bit QuickDraw and is included with the

system beginning with system software version 7.0), then the `ctSeed` field should be given a new value with another call to `_GetCTSeed`.

`ctFlags` indicates the Boolean characteristics of a color table. If the most significant bit of `ctFlags` is clear, then the `value` field of each `ColorSpec` entry in the `ctTable` array is interpreted as the pixel value for the color that's specified in the `rgb` field in the same `ColorSpec` entry. You can build a color table with nonconsecutive pixel values this way. If this bit is set, then all the `value` fields in the color table are ignored and the index of each `ColorSpec` record in the `ctTable` array is that record's pixel value. It's your choice whether to clear this bit and set the `value` fields or set this bit and ignore the `value` fields; traditionally this bit is clear for off-screen color tables.

If the next most significant bit of `ctFlags` is set, then the `value` field of each `ColorSpec` record in the `ctTable` array is used by `_CopyBits` as an index into the color palette that's attached to the destination window, and the `rgb` field is ignored. This is documented in the "Palette Manager" chapter of *Inside Macintosh* Volume VI, page 20-17.

The other bits are reserved for future use. If you create a color table from scratch, these other bits must be set to zero. If you use a color table that's generated by the system, then these bits must be preserved.

`ctSize` the number of color table entries minus 1. Normally, this field is set to 1, 3, 15, or 255 for 1-, 2-, 4-, and 8-bits per pixel, respectively. In special cases, it's reasonable to have less than the maximum number of entries for the pixel depth. For example, a color table for an 8-bit per pixel graphics environment could have just 150 entries, in which case the `ctSize` field should hold 149. For this case, it's still important to allocate as much space in the color table for the maximum number of entries for a pixel depth and clear the entries you're not using to zero because some parts of Color QuickDraw assume the size of a color table based on the pixel depth.

`ctTable` array of colors and pixel values. This table defines all the available colors in the color table and their pixel values. The `value` field of each `ColorSpec` record indicates that color's pixel value if the most significant bit of `ctFlags` is clear. It's ignored if the most significant bit of `ctFlags` is set. The `value` field is used as an index into a palette if the next most significant bit of `ctFlags` is set, in which case the `rgb` field is ignored. See the discussion of the `ctFlags` field earlier in this Note for more details.

Warning: Color QuickDraw's text-drawing routines assume that the color table of the destination graphics environment has the maximum number of colors for the pixel depth of the graphics environment, and that white is the first entry in the color table and black is the last entry. If these conditions aren't satisfied, then the resulting image is unpredictable.

The code fragment in Listing 6 shows how to allocate a 256-entry color table from scratch. Color tables have a variable size, so the `_NewHandle` call has to calculate the size of the `ColorTable` record plus the maximum number of color table entries for the pixel depth multiplied by the size of a `ColorSpec` record. `kNumColors - 1` is used in the calculation because the size of the `ColorTable` record includes the size of one `ColorSpec` entry in most development environments.

MPW Pascal Listing 6

```
CONST
   kNumColors = 256; {Number of color table entries}

VAR
   newColors: CTabHandle; {Handle to the new color table}
   index:     Integer;    {Index into the table of colors}

(* Allocate memory for the color table *)
newColors := CTabHandle(NewHandleClear(SizeOf (ColorTable) +
      SizeOf(ColorSpec) * (kNumColors - 1)));
IF newColors <> NIL THEN
   BEGIN
      (* Initialize the fields *)
      newColors^^.ctSeed := GetCTSeed;
      newColors^^.ctFlags := 0;
      newColors^^.ctSize := kNumColors - 1;

      (* Initialize the table of colors *)
      FOR index := 0 TO kNumColors - 1 DO
         BEGIN
            newColors^^.ctTable[index].value := index;
            newColors^^.ctTable[index].rgb.red := someRedValue;
            newColors^^.ctTable[index].rgb.green := someGreenValue;
            newColors^^.ctTable[index].rgb.blue := someBlueValue
         END
   END
```

MPW C Listing 6

```
#define kNumColors 256 /* Number of color table entries */

CTabHandle newColors; /* Handle to the new color table */
short      index;     /* Index into the table of colors */

/* Allocate memory for the color table */
newColors = (CTabHandle)NewHandleClear( sizeof (ColorTable) +
        sizeof (ColorSpec) * (kNumColors - 1) );
if (newColors != nil)
{
    /* Initialize the fields */
    (**newColors).ctSeed = GetCTSeed();
    (**newColors).ctFlags = 0;
    (**newColors).ctSize = kNumColors - 1;

    /* Initialize the table of colors */
    for (index = 0; index < kNumColors; index++)
    {
        (**newColors).ctTable[index].value = index;
        (**newColors).ctTable[index].rgb.red = someRedValue;
        (**newColors).ctTable[index].rgb.green = someGreenValue;
        (**newColors).ctTable[index].rgb.blue = someBlueValue;
    }
}
```

Changing Your Environment

After you create an off-screen graphics environment with certain dimensions, you might later want to change its size, depth, or color table without creating a completely new graphics environment from scratch and without needing to redraw the existing image. The UpdateOffScreen routine in

Listing 7 shows just one way to do this. It takes the same parameters that CreateOffScreen (defined in Listing 1) does, but instead of creating a new `CGrafPort` and `GDevice`, it alters the ones that you pass through the `updPort` and `updGDevice` parameters. If the `newBounds` parameter specifies an empty rectangle, then the existing boundary rectangle for the off-screen graphics environment is used. Similarly, if `newDepth` is zero, then the existing depth is used; and if the `newColors` parameter is NIL, then the existing color table is used. UpdateOffScreen alters the given `CGrafPort` and `GDevice` to the new settings, but it completely replaces the `PixMap`. After all the alterations are made, the old `PixMap`'s image is copied to the new `PixMap`'s image, and then the old `PixMap` and its image are disposed.

MPW Pascal Listing 7

```
FUNCTION UpdateOffScreen(
   newBounds:  Rect;       {New bounding rectangle of off-screen}
   newDepth:   Integer;    {New number of bits per pixel in off-screen}
   newColors:  CTabHandle; {New color table to assign to off-screen}
   updPort:    CGrafPtr;   {Returns a pointer to the updated CGrafPort}
   updGDevice: GDHandle    {Returns a handle to the updated GDevice}
   ): OSErr;

   CONST
      kMaxRowBytes = $3FFE; {Maximum number of bytes per row of pixels}

   VAR
      newPixMap:   PixMapHandle; {Handle to the new off-screen PixMap}
      oldPixMap:   PixMapHandle; {Handle to the old off-screen PixMap}
      bounds:      Rect;         {Boundary rectangle of off-screen}
      depth:       Integer;      {Depth of the off-screen PixMap}
      bytesPerRow: Integer;      {Number of bytes per row in the PixMap}
      colors:      CTabHandle;   {Colors for the off-screen PixMap}
      savedFore:   RGBColor;     {Saved foreground color}
      savedBack:   RGBColor;     {Saved background color}
      aColor:      RGBColor;     {Used to set foreground and background color}
      qdVersion:   LongInt;      {Version of QuickDraw currently in use}
      savedPort:   GrafPtr;      {Pointer to GrafPort used for save/restore}
      savedDevice: GDHandle;     {Handle to GDevice used for save/restore}
      savedState:  SignedByte;   {Saved state of color table handle}
      error:       OSErr;        {Returns error code}

BEGIN
   (* Initialize a few things before we begin *)
   newPixMap := NIL;
   error := noErr;

   (* Keep the old bounds rectangle, or get the new one *)
   IF EmptyRect(newBounds) THEN
      bounds := updPort^.portRect
   ELSE
      bounds := newBounds;

   (* Keep the old depth, or get the old one *)
   IF newDepth = 0 THEN
      depth := updPort^.portPixMap^^.pixelSize
   ELSE
      depth := newDepth;

   (* Get the old clut, or save new clut's state and make it nonpurgeable *)
   IF newColors = NIL THEN
      colors := updPort^.portPixMap^^.pmTable
   ELSE
      BEGIN
         savedState := HGetState(Handle(newColors));
```

```
      HNoPurge(Handle(newColors));
      colors := newColors;
   END;

(* Calculate the number of bytes per row in the off-screen PixMap *)
bytesPerRow := ((depth * (bounds.right - bounds.left) + 31) DIV 32) * 4;

(* Get the current QuickDraw version *)
error := Gestalt (gestaltQuickdrawVersion, qdVersion);
error := noErr;

(* Make sure depth is indexed or depth is direct and 32-Bit QD installed *)
IF (depth = 1) OR (depth = 2) OR (depth = 4) OR (depth = 8) OR
      (((depth = 16) OR (depth = 32)) AND (qdVersion >= gestalt32BitQD)) THEN
   BEGIN
      (* Maximum number of bytes per row is 16,382; make sure within range *)
      IF bytesPerRow <= kMaxRowBytes THEN
         BEGIN
            (* Make sure a color table is provided if the depth is indexed *)
            IF depth <= 8 THEN
               IF colors = NIL THEN
                  (* Indexed depth and clut is NIL; is parameter error *)
                  error := paramErr;
         END
      ELSE
         (* # of bytes per row is more than 16,382; is parameter error *)
         error := paramErr;
   END
ELSE
   (* Pixel depth isn't valid; is parameter error *)
   error := paramErr;

(* If sanity checks succeed, attempt to update the graphics environment *)
IF error = noErr THEN
   BEGIN
      (* Allocate a new PixMap *)
      newPixMap := PixMapHandle(NewHandleClear(SizeOf(PixMap)));
      IF newPixMap <> NIL THEN
           BEGIN
            (* Initialize the new PixMap for off-screen drawing *)
            error := SetUpPixMap(depth, bounds, colors, bytesPerRow,
                  newPixMap);
            IF error = noErr THEN
               BEGIN
                  (* Save old PixMap and install new, initialized one *)
                  oldPixMap := updPort^.portPixMap;
                  updPort^.portPixMap := newPixMap;

                  (* Save current port & GDevice; set ones we're updating *)
                  GetPort(savedPort);
                  savedDevice := GetGDevice;
                  SetPort(GrafPtr(updPort));
                  SetGDevice(updGDevice);

                  (* Set portRect, visRgn, clipRgn to given bounds rect *)
                  updPort^.portRect := bounds;
                  RectRgn(updPort^.visRgn, bounds);
                  ClipRect(bounds);

                  (* Update the GDevice *)
                  IF newPixMap^^.pixelSize <= 8 THEN
                     updGDevice^^.gdType := clutType
                  ELSE
```

```
                        updGDevice^^.gdType := directType;
                    updGDevice^^.gdPMap := newPixMap;
                    updGDevice^^.gdRect := newPixMap^^.bounds;

                    (* Set color-device bit if PixMap isn't black & white *)
                    IF newPixMap^^.pixelSize > 1 THEN
                        SetDeviceAttribute(updGDevice, gdDevType, TRUE);
                    else
                        SetDeviceAttribute(updGDevice, gdDevType, FALSE);

                    (* Save current fore/back colors and set to B&W *)
                    GetForeColor(savedFore);
                    GetBackColor(savedBack);
                    aColor.red := 0; aColor.green := 0; aColor.blue := 0;
                    RGBForeColor(aColor);
                    aColor.red := $FFFF;
                    aColor.green := $FFFF;
                    aColor.blue := $FFFF;
                    RGBBackColor(aColor);

                    (* Copy old image to the new graphics environment *)
                    HLock(Handle(oldPixMap));
                    CopyBits(BitMapPtr(oldPixMap^)^, GrafPtr(updPort)^.portBits,
                          oldPixMap^^.bounds, updPort^.portRect,
                          srcCopy, NIL);
                    HUnlock(Handle(oldPixMap));

                    (* Restore the foreground/background color *)
                    RGBForeColor(savedFore);
                    RGBBackColor(savedBack);

                    (* Restore the saved port *)
                    SetPort(savedPort);
                    SetGDevice(savedDevice);

                    (* Get rid of the old PixMap and its dependents *)
                    DisposPtr(oldPixMap^^.baseAddr);
                    DisposeCTable(oldPixMap^^.pmTable);
                    DisposHandle(Handle(oldPixMap));
                END;
            END
        ELSE
            error := MemError;
    END;

    (* Restore the given state of the color table *)
    IF colors <> NIL THEN
        HSetState(Handle(colors), savedState);

    (* One Last Look Around The House Before We Go… *)
    IF error <> noErr THEN
        BEGIN
            IF newPixMap <> NIL THEN
                BEGIN
                    IF newPixMap^^.pmTable <> NIL THEN
                        DisposCTable(newPixMap^^.pmTable);
                    IF newPixMap^^.baseAddr <> NIL THEN
                        DisposPtr(newPixMap^^.baseAddr);
                    DisposHandle(Handle(newPixMap));
                END;
        END;
    UpdateOffScreen := error;
END;
```

MPW C Listing 7

```
#define kMaxRowBytes 0x3FFE /* Maximum number of bytes in a row of pixels */

OSErr UpdateOffScreen(
    Rect       *newBounds, /* New bounding rectangle of off-screen */
    short      newDepth,   /* New number of bits per pixel in off-screen */
    CTabHandle newColors,  /* New color table to assign to off-screen */
    CGrafPtr   updPort,    /* Returns a pointer to the updated CGrafPort */
    GDHandle   updGDevice) /* Returns a handle to the updated GDevice */
{
    PixMapHandle newPixMap;   /* Handle to the new off-screen PixMap */
    PixMapHandle oldPixMap;   /* Handle to the old off-screen PixMap */
    Rect         bounds;      /* Boundary rectangle of off-screen */
    short        depth;       /* Depth of the off-screen PixMap */
    short        bytesPerRow; /* Number of bytes per row in the PixMap */
    CTabHandle   colors;      /* Colors for the off-screen PixMap */
    RGBColor     savedFore;   /* Saved foreground color */
    RGBColor     savedBack;   /* Saved background color */
    RGBColor     aColor;      /* Used to set foreground and background color */
    long         qdVersion;   /* Version of QuickDraw currently in use */
    GrafPtr      savedPort;   /* Pointer to GrafPort used for save/restore */
    GDHandle     savedDevice; /* Handle to GDevice used for save/restore */
    SignedByte   savedState;  /* Saved state of color table handle */
    OSErr        error;       /* Returns error code */

    /* Initialize a few things before we begin */
    newPixMap = nil;
    error = noErr;

    /* Keep the old bounds rectangle, or get the new one */
    if (EmptyRect( newBounds ))
        bounds = updPort->portRect;
    else
        bounds = *newBounds;

    /* Keep the old depth, or get the old one */
    if (newDepth == 0)
        depth = (**updPort->portPixMap).pixelSize;
    else
        depth = newDepth;

    /* Get the old clut, or save new clut's state and make it nonpurgeable */
    if (newColors == nil)
        colors = (**updPort->portPixMap).pmTable;
    else
    {
        savedState = HGetState( (Handle)newColors );
        HNoPurge( (Handle)newColors );
        colors = newColors;
    }

    /* Calculate the number of bytes per row in the off-screen PixMap */
    bytesPerRow = ((depth * (bounds.right - bounds.left) + 31) >> 5) << 2;

    /* Get the current QuickDraw version */
    (void)Gestalt( gestaltQuickdrawVersion, &qdVersion );

    /* Make sure depth is indexed or depth is direct and 32-Bit QD installed */
    if (depth == 1 || depth == 2 || depth == 4 || depth == 8 ||
            ((depth == 16 || depth == 32) && qdVersion >= gestalt32BitQD))
    {
        /* Maximum number of bytes per row is 16,382; make sure within range */
```

```
        if (bytesPerRow <= kMaxRowBytes)
        {
            /* Make sure a color table is provided if the depth is indexed */
            if (depth <= 8)
                if (colors == nil)
                    /* Indexed depth and clut is NIL; is parameter error */
                    error = paramErr;
        }
        else
            /* # of bytes per row is more than 16,382; is parameter error */
            error = paramErr;
    }
    else
        /* Pixel depth isn't valid; is parameter error */
        error = paramErr;

    /* If sanity checks succeed, attempt to create a new graphics environment */
    if (error == noErr)
    {
        /* Allocate a new PixMap */
        newPixMap = (PixMapHandle)NewHandleClear( sizeof (PixMap) );
        if (newPixMap != nil)
        {
            /* Initialize the new PixMap for off-screen drawing */
            error = SetUpPixMap( depth, &bounds, colors, bytesPerRow, newPixMap );
            if (error == noErr)
            {
                /* Save the old PixMap and install the new, initialized one */
                oldPixMap = updPort->portPixMap;
                updPort->portPixMap = newPixMap;

                /* Save current port & GDevice and set ones we're updating */
                GetPort( &savedPort );
                savedDevice = GetGDevice();
                SetPort( (GrafPtr)updPort );
                SetGDevice( updGDevice );

                /* Set portRect, visRgn, and clipRgn to the given bounds rect */
                updPort->portRect = bounds;
                RectRgn( updPort->visRgn, &bounds );
                ClipRect( &bounds );

                /* Update the GDevice */
                if ((**newPixMap).pixelSize <= 8)
                    (**updGDevice).gdType = clutType;
                else
                    (**updGDevice).gdType = directType;
                (**updGDevice).gdPMap = newPixMap;
                (**updGDevice).gdRect = (**newPixMap).bounds;

                /* Set color-device bit if PixMap isn't black & white */
                if ((**newPixMap).pixelSize > 1)
                    SetDeviceAttribute( updGDevice, gdDevType, true );
                else
                    SetDeviceAttribute( updGDevice, gdDevType, false );

                /* Save current foreground/background colors and set to B&W */
                GetForeColor( &savedFore );
                GetBackColor( &savedBack );
                aColor.red = aColor.green = aColor.blue = 0;
                RGBForeColor( &aColor );
                aColor.red = aColor.green = aColor.blue = 0xFFFF;
                RGBBackColor( &aColor );

                /* Copy old image to the new graphics environment */
```

```
                HLock( (Handle)oldPixMap );
                CopyBits( (BitMapPtr)*oldPixMap, &((GrafPtr) updPort)->portBits,
                        &(**oldPixMap).bounds, &updPort->portRect,
                        srcCopy, nil );
                HUnlock( (Handle)oldPixMap );

                /* Restore the foreground/background color */
                RGBForeColor( &savedFore );
                RGBBackColor( &savedBack );

                /* Restore the saved port */
                SetPort( savedPort );
                SetGDevice( savedDevice );

                /* Get rid of the old PixMap and its dependents */
                DisposPtr( (**oldPixMap).baseAddr );
                DisposeCTable( (**oldPixMap).pmTable ) ;
                DisposHandle( (Handle)oldPixMap );
            }
        }
        else
            error = MemError();
    }

    /* Restore the given state of the color table */
    if (colors != nil)
        HSetState( (Handle)colors, savedState );

    /* One Last Look Around The House Before We Go… */
    if (error != noErr)
    {
        /* Some error occurred; dispose of everything we allocated */
        if (newPixMap != nil)
        {
            if ((**newPixMap).pmTable)
                DisposCTable( (**newPixMap).pmTable );
            if ((**newPixMap).baseAddr)
                DisposPtr ( (**newPixMap).baseAddr );
            DisposHandle( (Handle)newPixMap );
        }
    }
    return error;
}
```

UpdateOffScreen begins by checking the boundary rectangle, depth, or color table for emptiness, zero, or NIL, respectively. If any these satisfy that condition, then the existing characteristic is used. Next, the same sanity check that CreateOffScreen uses is done. If this sanity check succeeds, then a new `PixMap` is allocated, and then it's initialized by the SetUpPixMap routine that's given in Listing 2 which gives the new `PixMap` a new pixel image and its own copy of the color table. This new `PixMap` is installed into the `CGrafPort` after saving the reference to the old `PixMap`. Then, the `portRect`, `visRgn`, and `clipRgn` of the `CGrafPort` are set to the new boundary rectangle, as is the `gdRect` of the `GDevice`. The `gdType` of the `GDevice` is set either for the indexed-color or direct-color model, the `gdPMap` is set to the new `PixMap`, and the device attributes are set according to the pixel depth. Details about the settings for the `CGrafPort` and `GDevice` are in "Building the CGrafPort" and "Building the GDevice," respectively, earlier in this Note.

At this point, the off-screen graphics environment is ready with its new characteristics, but it has garbage for an image because nothing has been drawn into it yet. The old `PixMap`, pixel image, and color table are still around, so `_CopyBits` transfers the old image into the altered graphics environment. `_CopyBits` handles the mapping from the old image's characteristics to the new

characteristics, so the altered graphics environment gets the best possible representation of the old image according to its new characteristics.

Changing the Off-Screen Color Table

Sometimes, it's useful to change some or all of the colors in an off-screen color table, or to replace the off-screen color table with another one, so that the existing image in an indexed-color graphics environment appears with new colors. For example, if you had an off-screen image of a blue car and wanted to see what it looked like in green, you could change all of the shades of blue in the off-screen color table to green, and then `_CopyBits` the image to the screen. Notice that this is different from calling the UpdateOffScreen routine in the previous section with a different color table. That routine tries to reproduce the colors from the original image as best it can in the new set of colors. This section discusses the case in which you want the image's colors to change.

The most obvious part of doing this is simply to get the color table from the off-screen pixel map's `pmTable` field and modify the entries, or to dispose of the off-screen graphics environment's current color table and assign the new one to it. There's one more step to complete the process though. The discussion about `GDevice` records in "The Building Blocks" in this Note discusses inverse tables and how they go hand-in-hand with color tables. If you alter or replace the color table, you have to make sure that the inverse table of the off-screen drawing environment is rebuilt according to the new colors because Color QuickDraw uses that inverse table to know what pixel values to use for the specified color. You don't have to rebuild the inverse table explicitly as long as you tell Color QuickDraw that the color table changed. To do this, all you have to do is make sure that the `ctSeed` of the changed or altered color table is set to a new value. And to do this, you can simply call `_CTabChanged`, which is documented on page 17-26 of the "Color QuickDraw" chapter of *Inside Macintosh* Volume VI. `_CTabChanged` is available beginning with 32-Bit QuickDraw and it's available in system software version 7.0. If this routine isn't available, then you can still tell Color QuickDraw that the color table has been changed by calling `_GetCTSeed` and assigning its result directly to your new color table's `ctSeed` field.

The next time you draw into this off-screen drawing environment, Color QuickDraw checks the `ctSeed` of the environment's color table against the `iTabSeed` of the inverse table of the environment's `GDevice`. Because you changed the `ctSeed` of the color table either through `_CTabChanged` or `_GetCTSeed`, these two seeds are different so Color QuickDraw automatically rebuilds the inverse table of the current `GDevice` and then it copies the `ctSeed` of the color table to the `iTabSeed` of the rebuilt inverse table. Then drawing continues normally.

Follow That Screen!

One common need of off-screen graphics environments is that they have a depth and color table that matches a screen. The CreateOffScreen routine requires a color table for indexed-color environments, and a pixel depth. Because there can be more than one screen attached to a Macintosh system, you have to decide which screen's depth and color table you should use. Typically, the depth and color table of the deepest screen that contains the area that you're interested in (probably the area of a window) is used. Another option is to use the depth and color table of the screen that has the largest area of intersection with the area that you're interested in. To find the depth and color table of the screen on which you want to base an off-screen graphics environment, you must use the list of graphics devices for all screens which is maintained by the system. Every `GDevice` record for a screen has a handle to that screen's `PixMap`, and you can find the screen's depth and color table there.

Listing 8 shows a routine called CreateScreenOffScreen which creates an off-screen graphics environment that has the depth and color table of a selected screen. The first parameter, `bounds`, specifies the rectangular part of the screen area in which you're interested in global coordinates.

The `screenOption` parameter specifies how you want the screen to be chosen. If you pass `kDeepestScreen` in this parameter, CreateScreenOffScreen creates the new off-screen graphics environment with the depth and color table of the deepest screen that intersects the `bounds` rectangle. If you instead pass `kLargestScreenArea`, then the new off-screen graphics environment is created with the depth and color table of the screen with the largest area of intersection with the `bounds` rectangle.

MPW Pascal Listing 8

```
TYPE
   ScreenOpt = (kDeepestScreen, kLargestAreaScreen);

FUNCTION CreateScreenOffScreen(
   bounds:          Rect;      {Global rectangle of part of screen to save}
   screenOption:    ScreenOpt; {Use deepest or largest intersection area screen?}
   VAR retPort:     CGrafPtr;  {Returns a pointer to the new CGrafPort}
   VAR retGDevice: GDHandle    {Returns a handle to the new GDevice}
   ): OSErr;

   VAR
      baseGDevice:  GDHandle;     {GDevice to base off-screen on}
      aGDevice:     GDHandle;     {Handle to each GDevice in the GDevice list}
      basePixMap:   PixMapHandle; {baseGDevice's PixMap}
      maxArea:      LongInt;      {Largest intersection area found}
      area:         LongInt;      {Area of rectangle of intersection}
      commonRect:   Rect;         {Rectangle of intersection}
      normalBounds: Rect;         {bounds rectangle normalized to (0, 0)}
      error:        Integer;      {Error code}

BEGIN
   error := noErr;

   (* Different screen options require different algorithms *)
   IF screenOption = kDeepestScreen THEN
      (* Graphics Devices Manager tells us the deepest intersecting screen *)
      baseGDevice := GetMaxDevice(bounds)
   ELSE IF screenOption = kLargestAreaScreen THEN
      BEGIN
         (* Get a handle to the first GDevice in the GDevice list *)
         aGDevice := GetDeviceList;

         (* Keep looping until all GDevices have been checked *)
         maxArea := 0;
         baseGDevice := NIL;
         WHILE aGDevice <> NIL DO
            BEGIN
               (* Check to see whether screen rectangle and bounds intersect *)
               IF SectRect(aGDevice^^.gdRect, bounds, commonRect) THEN
                  BEGIN
                     (* Calculate area of intersection *)
                     area := LongInt(commonRect.bottom - commonRect.top) *
                           LongInt(commonRect.right - commonRect.left);

                     (* Keep track of largest area of intersection so far *)
                     IF area > maxArea THEN
                        BEGIN
                           maxArea := area;
                           baseGDevice := aGDevice;
                        END;
                  END;

               (* Go to the next GDevice in the GDevice list *)
```

```
            aGDevice := GetNextDevice(aGDevice);
         END;
      END
   ELSE
      error := paramErr;

   (* If no screens intersect the bounds, baseDevice is NIL *)
   IF (baseGDevice <> NIL) AND (error = noErr) THEN
      BEGIN
         (* Normalize the bounds rectangle *)
         normalBounds := bounds;
         OffsetRect(normalBounds, -normalBounds.left, -normalBounds.top);

         (* Create off-screen graphics environment w/ depth, clut of screen *)
         basePixMap := baseGDevice^^.gdPMap;
         error := CreateOffScreen(normalBounds, basePixMap^^.pixelSize,
               basePixMap^^.pmTable, retPort, retGDevice);
      END;
   CreateScreenOffScreen := error;
END;
```

MPW C Listing 8

```
enum
{
    kDeepestScreen,
    kLargestAreaScreen,
};

OSErr CreateScreenOffScreen(
    Rect     *bounds,      /* Global rectangle of part of screen to save */
    short    screenOption, /* Use deepest or largest intersection area screen */
    CGrafPtr *retPort,     /* Returns a pointer to the new CGrafPort */
    GDHandle *retGDevice)  /* Returns a handle to the new GDevice */
{
    GDHandle     baseGDevice;  /* GDevice to base off-screen on */
    GDHandle     aGDevice;     /* Handle to each GDevice in the GDevice list */
    PixMapHandle basePixMap;   /* baseGDevice's PixMap */
    long         maxArea;      /* Largest intersection area found */
    long         area;         /* Area of rectangle of intersection */
    Rect         commonRect;   /* Rectangle of intersection */
    Rect         normalBounds; /* bounds rectangle normalized to (0, 0) */
    short        error;        /* Error code */

    error = noErr;

    /* Different screen options require different algorithms */
    if (screenOption == kDeepestScreen)
        /* Graphics Devices Manager tells us the deepest intersecting screen */
        baseGDevice = GetMaxDevice( bounds );
    else if (screenOption == kLargestAreaScreen)
    {
        /* Get a handle to the first GDevice in the GDevice list */
        aGDevice = GetDeviceList();

        /* Keep looping until all GDevices have been checked */
        maxArea = 0;
        baseGDevice = nil;
        while (aGDevice != nil)
        {
            /* Check to see whether screen rectangle and bounds intersect */
            if (SectRect( &(**aGDevice).gdRect, bounds, &commonRect ))
            {
                /* Calculate area of intersection */
```

```
                area = (long)(commonRect.bottom - commonRect.top) *
                       (long)(commonRect.right - commonRect.left);

                /* Keep track of largest area of intersection found so far */
                if (area > maxArea)
                {
                    maxArea = area;
                    baseGDevice = aGDevice;
                }
            }

            /* Go to the next GDevice in the GDevice list */
            aGDevice = GetNextDevice( aGDevice );
        }
    }
    else
        error = paramErr;

    /* If no screens intersect the bounds, baseDevice is NIL */
    if (baseGDevice != nil && error == noErr)
    {
        /* Normalize the bounds rectangle */
        normalBounds = *bounds;
        OffsetRect( &normalBounds, -normalBounds.left, -normalBounds.top );

        /* Create off-screen graphics environment w/ depth, clut of screen */
        basePixMap = (**baseGDevice).gdPMap;
        error = CreateOffScreen( &normalBounds, (**basePixMap).pixelSize,
                (**basePixMap).pmTable, retPort, retGDevice );
    }
    return error;
}
```

Finding the deepest screen that intersects an on-screen area is trivially easy because there's a Graphics Devices Manager routine that finds it called `_GetMaxDevice` which is documented on page 21-22 of the "Graphics Devices Manager" chapter of *Inside Macintosh* Volume VI. The rectangle in global coordinates of the screen area you're interested in is passed to `_GetMaxDevice`, and it returns a handle to the deepest screen that intersects that area, even if the area of intersection is as small as one pixel. If no screens intersect that area, then `_GetMaxDevice` returns NIL.

Finding the `GDevice` of the screen that has the maximum area of intersection with the screen area you're interested in isn't quite so easy because there's no single Graphics Devices Manager routine to find this `GDevice`; you have to search the `GDevice` list yourself. You can get a handle to the first `GDevice` in the list by calling `_GetDeviceList`, and you can get a handle to each successive `GDevice` by calling `_GetNextDevice`. `_GetDeviceList` is documented on pages 21-21 through 21-22 of the "Graphics Devices Manager" chapter of *Inside Macintosh* Volume VI, and `_GetNextDevice` is documented on page 21-22 of the same chapter. For each `GDevice` in the list, the area of intersection between the `bounds` and the `gdRect` of the `GDevice` is calculated. If the calculated area is the largest area of intersection found so far, then that area and the `GDevice` of that screen are remembered.

Once a winning `GDevice` has been chosen, either by being the deepest intersecting `GDevice` or the `GDevice` with the largest intersecting area, then CreateOffScreen routine is called with the pixel depth and color table of the `PixMap` of the `GDevice`, and the `bounds` rectangle normalized so that its top-left coordinate has the coordinates (0, 0). CreateOffScreen returns with the new off-screen graphics environment, and CreateScreenOffScreen returns this to the caller.

Choosing Your Off-Screen Memory

The CreateOffScreen routine in Listing 1 creates an off-screen graphics environment with its pixel image allocated as a nonrelocatable block in the application's heap. But this isn't the only way that the pixel image can be allocated. Pixel images can be big, and big blocks of nonrelocatable memory in your heap can be expensive in terms of performance, and they can cause a bad case of heap fragmentation. Why not put the pixel image in a relocatable block of memory instead? If there isn't much free memory in your heap and if MultiFinder or system software version 7.0 is running, there's memory that's not being used by any open applications, called *temporary memory* (formerly called *MultiFinder temporary memory*). Why not use this area of memory for the pixel image? Some people have NuBus cards with plenty of memory on them. Why not move the pixel image out of the heaps altogether and instead use NuBus memory for the pixel image? All of these things can be done with simple modifications to what's been discussed in this Note, and these modifications are discussed in the next few paragraphs.

How can pixel images be relocatable? After all, pixel images are referred to only by the `baseAddr` field of a `PixMap`, and the `baseAddr` is a pointer, not a handle. It's true that while QuickDraw is being used to draw into a graphics environment, the pixel image had better not move or else QuickDraw will start drawing over the area of memory that the pixel image used to be rather than where it is. But if QuickDraw isn't doing anything with the graphics environment, then it doesn't care what happens to the pixel image as long as the `baseAddr` points to it once QuickDraw starts drawing into the graphics environment. This implies a strategy: allocate the pixel image as a relocatable block and let it float in the heap; when QuickDraw is about to to draw into the graphics environment or to copy from it, lock the pixel image and copy its master pointer into the `baseAddr` field of the `PixMap`; when the drawing or copying is finished, unlock the pixel image. There are many ways to implement this, and Listing 9 shows a code fragment for one very simple method.

MPW Pascal Listing 9

```
      ...
      (* Allocate the pixel image; use long multiplication to avoid overflow *)
      offBaseAddr := NewHandle(LongInt(bytesPerRow) * (bounds^.bottom -
            bounds^.top));
      IF offBaseAddr <> NIL THEN
         BEGIN
            (* Initialize fields common to indexed and direct PixMaps *)
            aPixMap^^.baseAddr := Ptr(offBaseAddr); (* Reference the image *)
      ...


PROCEDURE LockOffScreen(
   offScreenPort: CGrafPtr {Ptr to off-screen CGrafPort}
   );

   VAR
      offImageHnd: Handle; {Handle to the off-screen pixel image}

BEGIN
   (* Get the saved handle to the off-screen pixel image *)
   offImageHnd := Handle(offScreenPort^.portPixMap^^.baseAddr);

   (* Lock the handle to the pixel image *)
   HLock(offImageHnd);

   (* Put pixel image master pointer into baseAddr so that QuickDraw can use it *)
   offScreenPort^.portPixMap^^.baseAddr := offImageHnd^;
END;
```

```
PROCEDURE UnlockOffScreen(
   offScreenPort: CGrafPtr {Ptr to off-screen port}
   );

   VAR
      offImagePtr: Ptr;    {Pointer to the off-screen pixel image}
      offImageHnd: Handle; {Handle to the off-screen pixel image}

BEGIN
   (* Get the handle to the off-screen pixel image *)
   offImagePtr := offScreenPort^.portPixMap^^.baseAddr;
   offImageHnd := RecoverHandle(offImagePtr);

   (* Unlock the handle *)
   HUnlock(offImageHnd);

   (* Save the handle back in the baseAddr field *)
   offScreenPort^.portPixMap^^.baseAddr := Ptr(offImageHnd);
END;
```

MPW C Listing 9

```
        ...
        /* Allocate the pixel image; use long multiplication to avoid overflow */
        offBaseAddr = NewHandle( (unsigned long)bytesPerRow * (bounds->bottom -
                bounds->top) );
        if (offBaseAddr != nil)
        {
            /* Initialize fields common to indexed and direct PixMaps */
            (**aPixMap).baseAddr = (Ptr)offBaseAddr; /* Reference the image */
        ...


void LockOffScreen(
    CGrafPtr offScreenPort) /* Pointer to the off-screen CGrafPort */
{
    Handle offImageHnd; /* Handle to the off-screen pixel image */

    /* Get the saved handle to the off-screen pixel image */
    offImageHnd = (Handle)(**offScreenPort->portPixMap).baseAddr;

    /* Lock the handle to the pixel image */
    HLock( offImageHnd );

    /* Put pixel image master pointer into baseAddr so that QuickDraw can use it */
    (**offScreenPort->portPixMap).baseAddr = *offImageHnd;
}


void UnlockOffScreen(
    CGrafPtr offScreenPort) /* Pointer to the off-screen CGrafPort */
{
    Ptr    offImagePtr; /* Pointer to the off-screen pixel image */
    Handle offImageHnd; /* Handle to the off-screen pixel image */

    /* Get the handle to the off-screen pixel image */
    offImagePtr = (**offScreenPort->portPixMap).baseAddr;
    offImageHnd = RecoverHandle( offImagePtr );

    /* Unlock the handle */
    HUnlock( offImageHnd );

    /* Save the handle back in the baseAddr field */
```

```
    (**offScreenPort->portPixMap).baseAddr = (Ptr)offImageHnd;
}
```

Listing 9 starts with a code fragment from the SetUpPixMap routine that's modified so that it allocates a new handle for the off-screen pixel image instead of a new pointer. This handle is saved in the `baseAddr` field for now. When you're about to draw into the off-screen graphics environment or to copy from it, the LockOffScreen routine in Listing 9 should be called with a pointer to the off-screen graphics environment's `CGrafPort` as the parameter. It takes the handle to the pixel image from the `baseAddr` field of the off-screen graphics environment's `PixMap` and passes it to `_HLock` which makes sure the pixel image can't move in the heap. Then, the pixel image's handle is dereferenced to get the master pointer to the pixel image, and this master pointer is copied into the `baseAddr` field. Now, QuickDraw can draw into or copy from the off-screen graphics environment.

When you're finished drawing into the off-screen graphics environment, the pixel image should be unlocked, and the UnlockOffScreen routine in Listing 9 does this. The `baseAddr` field of the `PixMap` holds the pixel image's master pointer, so this is passed to `_RecoverHandle` to get the pixel image's handle. This handle is passed to `_HUnlock` to let the pixel image float in the heap again, and then this handle is saved in the `baseAddr` field.

One potentially useful addition to the LockOffScreen routine would be a call to `_MoveHHi` just before the call to `_HLock`. This helps reduce heap fragmentation while the pixel image is locked by moving it up as high in the heap as possible before locking it, allowing the other relocatable blocks to move without tripping over it. You have to be careful with `_MoveHHi` though because it not only moves the handle as high in the heap as possible, it moves other relocatable blocks out of the top of the heap to make room for the handle. This could involve moving huge amounts of memory, and it's not unusual for `_MoveHHi` to take several seconds to do this.

How do you make an off-screen graphics environment that uses temporary memory for the pixel image? Temporary memory is allocated as handles, so there's almost no difference between using temporary memory and using relocatable blocks in your own heap in the way that Listing 9 shows. All you have to do is replace the calls to `_NewHandle`, `_HLock`, and `_HUnlock` with calls to `_TempNewHandle`, `_TempHLock`, and `_TempHUnlock`. If temporary memory handles are real, then you don't even have to replace the `_HLock` and `_HUnlock` calls—they work properly with temporary memory handles that are real.You can tell whether temporary memory handles are real or not by calling `_Gestalt` with the `gestaltOSAttr` selector. If the `gestaltRealTempMemory` bit is set, then all temporary memory handles are real. See the sections "About Temporary Memory" and "Using Temporary Memory" of *Inside Macintosh* Volume VI, pages 28-33 through 28-40.

How do you make an off-screen graphics environment that stores the pixel image on a NuBus memory card? The Macintosh Memory Manager doesn't keep track of heaps on NuBus memory cards so it can't be used to allocate memory on those cards, but if applications can use that card's memory at will, then an application can set up the off-screen graphics environment with its pixel image in the NuBus card's memory simply by setting the address of the card's memory in the `baseAddr` field of the off-screen graphics environment's `PixMap` instead of allocating anything.

If your NuBus memory card doesn't require 32-bit addressing mode to access its memory, then setting the `baseAddr` to the address of the NuBus card's memory is all you have to do. Some NuBus memory cards require its memory to be accessed in 32-bit addressing mode. Without 32-Bit QuickDraw, these memory cards can't be used for storing the pixel image of an off-screen graphics environment because Color QuickDraw without 32-Bit QuickDraw always reads and writes pixel images in 24-bit addressing mode regardless of whether the pixel image is in main memory, on a NuBus video card, or on a NuBus memory card. With 32-Bit QuickDraw, Color QuickDraw automatically switches to 32-bit addressing mode before reading or writing a pixel

image that's on a video card. It won't know to switch to 32-bit addressing mode if your off-screen graphics environment uses a pixel image on a NuBus memory card that's not a video card, but you can tell it to make this switch by setting bit 2 of the `pmVersion` field of the `PixMap` for the off-screen graphics environment. This is normally done by logically ORing the `pmVersion` field with the predefined constant `baseAddr32`. See "About 32-Bit Addressing" in Issue 6 of *develop*, page 36, for more details about how QuickDraw handles addressing modes.

The GWorld Factor

In May 1989, 32-Bit QuickDraw was introduced as an extension to the system. While it had a lot of new features, the GWorld mechanism was the one that made the big news. GWorlds are off-screen graphics environments that you can have the system put together in one call. There's no need for routines like CreateOffScreen, SetUpPixMap, or CreateGDevice—all of the off-screen graphics environment is set up with `_NewGWorld`. You can change most of its characteristics with `_UpdateGWorld`, set the current off-screen graphics environment with `_SetGWorld`, and get rid of the off-screen graphics environment with `_DisposeGWorld`. All the GWorld routines are described in the "Graphics Devices Manager" chapter of *Inside Macintosh* Volume VI. As an example, Listing 10 shows the same routine as the ExerciseOffScreen routine that's shown in Listing 5, but Listing 10 uses GWorlds rather than the do-it-yourself routines that are defined in this Note.

MPW Pascal Listing 10

```
PROCEDURE ExerciseOffScreen;

   CONST
      kOffDepth  = 8;    {Number of bits per pixel in off-screen environment}
      rGrayClut  = 1600; {Resource ID of gray-scale clut}
      rColorClut = 1601; {Resource ID of full-color clut}

   VAR
      grayPort:    GWorldPtr;  {Graphics environment for gray off screen}
      colorPort:   GWorldPtr;  {Graphics environment for color off screen}
      savedPort:   GrafPtr;    {Pointer to the saved graphics environment}
      savedDevice: GDHandle;   {Handle to the saved color environment}
      offColors:   CTabHandle; {Colors for off-screen environments}
      offRect:     Rect;       {Rectangle of off-screen environments}
      circleRect:  Rect;       {Rectangles for circle-drawing}
      count:       Integer;    {Generic counter}
      aColor:      RGBColor;   {Color used for drawing off-screen}
      error:       OSErr;      {Error return from off-screen creation}

BEGIN
   (* Set up the rectangle for the off-screen graphics environments *)
   SetRect(offRect, 0, 0, 256, 256);

   (* Get the color table for the gray off-screen graphics environment *)
   offColors := GetCTable(rGrayClut);

   (* Create the gray off-screen graphics environment *)
   error := NewGWorld(grayPort, kOffDepth, offRect, offColors, NIL, []);

   IF error = noErr THEN
      BEGIN
         (* Get the color table for the color off-screen graphics environment *)
         offColors := GetCTable(rColorClut);

         (* Create the color off-screen graphics environment *)
```

```
        error := NewGWorld(colorPort, kOffDepth, offRect, offColors, NIL, []);

        IF error = noErr THEN
           BEGIN
              (* Save the current graphics environment *)
              GetGWorld(savedPort, savedDevice);

              (* Set the current graphics environment to the gray one *)
              SetGWorld(grayPort, NIL);

              (* Draw gray-scale ramp into the gray off-screen environment *)
              FOR count := 0 TO 255 DO
                 BEGIN
                    aColor.red := count * 257;
                    aColor.green := aColor.red;
                    aColor.blue := aColor.green;
                    RGBForeColor(aColor);
                    MoveTo(0, count);
                    LineTo(255, count);
                 END;

              (* Copy gray ramp into color off-screen colorized with green *)
              SetGWorld(colorPort, NIL);
              aColor.red := $0000; aColor.green := $FFFF; aColor.blue := $0000;
              RGBForeColor(aColor);
              CopyBits(GrafPtr(grayPort)^.portBits,
                    GrafPtr(colorPort)^.portBits,
                    grayPort^.portRect,
                    colorPort^.portRect,
                    srcCopy + ditherCopy, NIL);

              (* Draw red, green, and blue circles *)
              PenSize(8, 8);
              aColor.red := $FFFF; aColor.green := $0000; aColor.blue := $0000;
              RGBForeColor(aColor);
              circleRect := colorPort^.portRect;
              FrameOval(circleRect);
              aColor.red := $0000; aColor.green := $FFFF; aColor.blue := $0000;
              RGBForeColor(aColor);
              InsetRect(circleRect, 20, 20);
              FrameOval(circleRect);
              aColor.red := $0000; aColor.green := $0000; aColor.blue := $FFFF;
              RGBForeColor(aColor);
              InsetRect(circleRect, 20, 20);
              FrameOval(circleRect);

              (* Copy the color off-screen environment to the current port *)
              SetGWorld(savedPort, savedDevice);
              CopyBits(GrafPtr(colorPort)^.portBits,
                    savedPort^.portBits,
                    colorPort^.portRect,
                    savedPort^.portRect,
                    srcCopy, NIL);

              (* Dispose of the off-screen graphics environments *)
              DisposeGWorld grayPort);
              DisposeGWorld(colorPort);
           END;
     END;
END;
```

MPW C Listing 10

```
#define kOffDepth  8    /* Number of bits per pixel in off-screen environment */
```

```
#define rGrayClut  1600 /* Resource ID of gray-scale clut */
#define rColorClut 1601 /* Resource ID of full-color clut */

void ExerciseOffScreen()
{
    GWorldPtr  grayPort;    /* Graphics environment for gray off screen */
    GWorldPtr  colorPort;   /* Graphics environment for color off screen */
    CGrafPtr   savedPort;   /* Pointer to the saved graphics environment */
    GDHandle   savedDevice; /* Handle to the saved color environment */
    CTabHandle offColors;   /* Colors for off-screen environments */
    Rect       offRect;     /* Rectangle of off-screen environments */
    Rect       circleRect;  /* Rectangles for circle-drawing */
    short      count;       /* Generic counter */
    RGBColor   aColor;      /* Color used for drawing off-screen */
    OSErr      error;       /* Error return from off-screen creation */

    /* Set up the rectangle for the off-screen graphics environments */
    SetRect( &offRect, 0, 0, 256, 256 );

    /* Get the color table for the gray off-screen graphics environment */
    offColors = GetCTable( rGrayClut );

    /* Create the gray off-screen graphics environment */
    error = NewGWorld( &grayPort, kOffDepth, &offRect, offColors, nil, 0 );

    if (error == noErr)
    {
        /* Get the color table for the color off-screen graphics environment */
        offColors = GetCTable( rColorClut );

        /* Create the color off-screen graphics environment */
        error = NewGWorld( &colorPort, kOffDepth, &offRect, offColors, nil, 0 );

        if (error == noErr)
        {
            /* Save the current graphics environment */
            GetGWorld( &savedPort, &savedDevice );

            /* Set the current graphics environment to the gray one */
            SetGWorld( grayPort, nil );

            /* Draw gray-scale ramp into the gray off-screen environment */
            for (count = 0; count < 256; count++)
            {
                aColor.red = aColor.green = aColor.blue = count * 257;
                RGBForeColor( &aColor );
                MoveTo( 0, count );
                LineTo( 255, count );
            }

            /* Copy gray ramp into color off-screen colorized with green */
            SetGWorld( colorPort, nil );
            aColor.red = 0x0000; aColor.green = 0xFFFF; aColor.blue = 0x0000;
            RGBForeColor( &aColor );
            CopyBits( &((GrafPtr)grayPort)->portBits,
                    &((GrafPtr)colorPort)->portBits,
                    &grayPort->portRect,
                    &colorPort->portRect,
                    srcCopy | ditherCopy, nil );

            /* Draw red, green, and blue circles */
            PenSize( 8, 8 );
            aColor.red = 0xFFFF; aColor.green = 0x0000; aColor.blue = 0x0000;
```

```
            RGBForeColor( &aColor );
            circleRect = colorPort->portRect;
            FrameOval( &circleRect );
            aColor.red = 0x0000; aColor.green = 0xFFFF; aColor.blue = 0x0000;
            RGBForeColor( &aColor );
            InsetRect( &circleRect, 20, 20 );
            FrameOval( &circleRect );
            aColor.red = 0x0000; aColor.green = 0x0000; aColor.blue = 0xFFFF;
            RGBForeColor( &aColor );
            InsetRect( &circleRect, 20, 20 );
            FrameOval( &circleRect );

            /* Copy the color off-screen environment to the current port */
            SetGWorld( savedPort, savedDevice );
            CopyBits( &((GrafPtr)colorPort)->portBits,
                    &((GrafPtr)savedPort)->portBits,
                    &colorPort->portRect,
                    &savedPort->portRect,
                    srcCopy, nil );

            /* Dispose of the off-screen graphics environments */
            DisposeGWorld( grayPort );
            DisposeGWorld( colorPort );
        }
    }
}
```

`_NewGWorld` creates an off-screen graphics environment by creating a `CGrafPort`, `PixMap`, and `GDevice`—the same structures that you normally put together when you make an off-screen graphics environment yourself. In this aspect, and in fact in most aspects, there's nothing magical about GWorlds. Do GWorlds make the CreateOffScreen, DisposeOffScreen, and their dependents useless? That depends on what your needs are. What follows are a few issues about off-screen drawing and how that determines whether you use your own routines, such as CreateOffScreen, to create and maintain off-screen graphics environments or whether you use GWorlds for the same purpose.

I Want the Best Performance!

As mentioned in the last paragraph, there's nothing magical about GWorlds in most aspects. In one major aspect, there certainly is: the version of Color QuickDraw that runs with the 8•24 GC video card's acceleration on knows about GWorlds and can cache their `CGrafPort`, `PixMap`, `GDevice`, inverse table, color table, and pixel image on the 8•24 GC card if there's enough memory on it. When this is done, QuickDraw operations on the GWorld can be much faster than they'd normally be because the image data can stay in the card's memory where the fast microprocessor is, and image data doesn't have to move across NuBus in transfer operations between the GWorld and the screen. Additionally, these operations are executed asynchronously which increases the overall speed of your programs. For details about how the 8•24 GC card and GC QuickDraw work, see Guillermo Ortiz's article, "Macintosh Display Card 8•24 GC: The Naked Truth," in Issue 5 of *develop*.

8•24 GC QuickDraw doesn't know about the off-screen graphics environments that you create, so it doesn't cache its structures. All QuickDraw commands that move image data between the off-screen graphics environment and the screen have to move the data across NuBus, and that slows down the operation in comparison to keeping all the image data on the card.

If you want the highest possible drawing and copying performance with the 8•24 GC card, you must use GWorlds for your off-screen graphics environments.

I Want to Use a NuBus Memory Card for My GWorld's Off-Screen Pixel Image

One common desire is to use a NuBus memory card to hold a pixel image. Because GWorlds are so easy to set up, and because GWorlds have all the same parts that you can make for an off-screen graphics environment, it's tempting to make a GWorld and then point the `baseAddr` of the GWorld's `PixMap` at the NuBus card's memory. But GWorlds are designed to be fairly atomic structures, so they can't be changed in this way. You can change a GWorld's dimensions, depth, and color table because there's a routine (`_UpdateGWorld`) that is designed to change these things, but you can't change the pixel image without risking future compatibility.

If you want to have an off-screen graphics environment use a NuBus video card to store the pixel image, you should set up your own off-screen graphics environment rather than use GWorlds. This is covered earlier in this Note in "Choosing Your Off-Screen Memory."

I Want My Program to Work on All System Software Releases

GWorlds have been around since 32-Bit QuickDraw was released (while system software version 6.0.3 was current). Until system software version 7.0, 32-Bit QuickDraw was an optional part of the system, so you aren't guaranteed use of GWorlds even under recent system software releases. Obviously, if GWorlds aren't available and your program still has to work with off-screen graphics environments, then there's no choice but to use your own routines for creating, maintaining, and disposing of off-screen graphics environments. What's usually done in these cases is to check via `_Gestalt` whether GWorlds are available or not. If they aren't, then you create your off-screen graphics environment with your own routines. If they are, then you can use GWorlds without having to take up memory with your code for creating off-screen graphics environments yourself.

Are We There Yet?

Reliable, understandable, and maintainable off-screen drawing routines means not taking short-cuts. The most common problems that people run into with off-screen drawing routines is the appearance of strange colors and the gradual degradation of reliability as the program does more off-screen drawing. Building an off-screen graphics environment out of a CGrafPort, GDevice, and PixMap or by using GWorlds, combined with an understanding of how Color QuickDraw uses off-screen graphics environments, helps get rid of these problems. Hopefully, this Note helps you understand these things so that you can get better programs out the door faster.

Further Reference:

- Apple Computer, Inc., *Inside Macintosh* Volume I, Addison-Wesley, Reading, MA, 1985
- Apple Computer, Inc., *Inside Macintosh* Volume V, Addison-Wesley, Reading, MA, 1988.
- Apple Computer, Inc., *Inside Macintosh* Volume VI, Addison-Wesley, Reading, MA, 1991.
- Knaster, S., *Macintosh Programming Secrets*, Addison-Wesley, Reading, MA, 1988.
- Leak, B., "Realistic Color For Real-World Applications," *develop*, January 1990, 4-21.
- Ortiz, G., "Braving Offscreen GWorlds," *develop*, January 1990, 28-40.

- Ortiz, G., "Deaccelerated _CopyBits & 8•24 GC QuickDraw," *Macintosh Technical Note #289*, January 1991.
- Ortiz, G., "Macintosh Display Card 8•24 GC: The Naked Truth," *develop*, July 1990, 332-347.
- Othmer, K., "QuickDraw's CopyBits Procedure: Better Than Ever in System 7.0," *develop*, Spring 1991, 23-42.
- Tanaka, F., "Of Time and Space and _CopyBits," *Macintosh Technical Note #277*, June 1990.
- Zap, J., F. Tanaka, J. Friedlander, and G. Jernigan, "Drawing Into an Off-Screen Bitmap," *Macintosh Technical Note #41*, June 1990.

NuBus is a trademark of Texas Instruments.

#121: Using the High-Level AppleTalk Routines

See also: The AppleTalk Manager
Inside AppleTalk
AppleTalk Manager Update

Written by: Fred A. Huxham May 4, 1987
Updated: March 1, 1988

What you need to do in order to use high-level AppleTalk routines depends upon the interfaces you are using. Some differences are outlined below.

MPW before 2.0

When calling the old high-level AppleTalk routines, many programmers get mysterious "resource not found" errors (-192) from such seemingly harmless routines as `MPPOpen`. The resource that is not being found is 'atpl', a resource that contains all the glue code to the high-level routines. In order to use the high-level routines, your application must have this resource in its resource fork. The 'atpl' resource is included in a file called "AppleTalk" with any compilers that use this outdated version of the AppleTalk interface.

MPW 2.0 and newer

A newer version of the alternate interfaces is available in MPW 2.0; it includes bug fixes and increased Macintosh II compatibility. With this version of the interface, the 'atpl' resource is no longer used. Glue code is now linked into your application.

This will be the final release of the current-style interface. It will be supported for some time as the **alternate interface**. We have moved to a more straightforward and simple **preferred interface**, which is also implemented in MPW 2.0 and newer, and is described in the AppleTalk Manager chapter of *Inside Macintosh* vol. V. Developers are free to continue to use the alternate interface, but in the long run it will be advantageous to move to the preferred interface.

Third Party Compilers

Third party compilers use interfaces that are built from Apple's MPW interfaces. Some compilers may not have upgraded to the new interfaces yet. Contact the individual compiler manufacturers for more information.

#122: Device-Independent Printing

See also: The Printing Manager

Written by: Ginger Jernigan
Updated:

May 4, 1987
March 1, 1988

The Printing Manager was designed to give Macintosh applications a device-independent method of printing, but we *have* provided device-dependent information, such as the contents of the print record. Due to the large number of printer-type drivers becoming available (even for non-printer devices) device independence is more necessary than ever. What this means to you, as a developer, is that we will no longer be providing (or supporting) information regarding the internal structure of the print record.

We realize that there are situations where the application may know the best method for printing a particular document and may want to bypass our dialogs. Unfortunately, using your own dialogs or not using the dialogs at all, requires setting the necessary fields in the print record yourself. There are a number of problems:

- Many of the fields in the print record are undocumented, and, as we change the internal architecture of the Printing Manager to accommodate new devices, those undocumented fields are likely to change.

- Each driver uses the private, and many of the public, fields in the print record differently. The implications are that you would need intimate knowledge of how each field is used by each available driver, and you would have to set the fields in the record *differently* depending on the driver chosen. As the number of available printer-type drivers increases, this can become a cumbersome task.

Summary

To be compatible with future printer-like devices, it is essential that your application print in a device-independent manner. Avoid testing undocumented fields, setting fields in the print record directly and bypassing the existing print dialogs. Use the Printing Manager dialogs, `PrintDefault` and `PrValidate` to set up the print record for you.

#123: Bugs in LaserWriter ROMs

See also: The Printing Manager
PostScript Language Reference Manual, Adobe Systems

Written by: Ginger Jernigan May 4, 1987
Modified by: Ginger Jernigan July 1, 1987
Updated: March 1, 1988

These are LaserWriter bugs that your users may encounter when printing from **any** Macintosh application. These are for your information; you cannot code around them. The bugs described here occur in the 1.0 and 2.0 LaserWriter ROMs.

To determine which ROMs their LaserWriter contains, users can look at the test page that the LaserWriter prints at start-up time. In addition to other information (detailed in the LaserWriter user's manual), the ROM version is shown at the bottom of the line graph. The original LaserWriter contained version 1.0 ROMs. The currently shipping LaserWriter and those upgraded to the LaserWriter Plus contain version 2.0 ROMs.

These are some of the problems we know of:

1. If the level of paper in the paper tray is getting low, and the user prints a document that will cause the tray to become empty, a PostScript error may occur. This problem exists in both the 1.0 and 2.0 LaserWriter ROMs and will **not** be fixed in the next ROM version.

2. If a user prints more than 15 copies of a document, a timeout condition may occur causing the print job to abort. With LaserShare, this problem can occur with as few as 9 copies. This problem is a result of the LaserWriter turning AppleTalk off while it is printing. It doesn't send out any packets to tell the world it's still alive while it is printing, so the connection times out after about 2 minutes. This problem exists in both the 1.0 and 2.0 LaserWriter ROMs and will **not** be fixed in the next ROM version.

3. When printing a document that contains more than 10 patterns, users may receive intermittent PostScript errors. This usually occurs when trying to print a lot of patterns, **and** a bitmap image on the same page. The code for imaging patterns allocates almost all of the available RAM for itself, so when the bitmap imaging code tries to allocate space, and there isn't enough (and it doesn't know how to reclaim memory from the previous operation), a `limitcheck` error occurs. This problem exists in 2.0 LaserWriter ROMs. It will be improved but **not** fixed in the next ROM version.

4. If a user chooses US Letter or B5 paper and has a different sized tray in the printer, and prints using manual feed, the LaserWriter will print assuming that the paper being fed manually is the same size as that in the tray. For example, if they have a US letter tray in the LaserWriter and print a document formatted for B5 letter using manual feed, the image will not be centered on the page. The printer assumes that the manually fed paper is also US letter size and prints the image positioned accordingly, despite the driver's instructions. This is a bug in the `Note` operator in PostScript, which the driver uses for specifying the US letter and B5 letter paper sizes. The workaround is to tell the user to put an B5 tray in the printer when printing B5 manually. This problem exists in the 1.0 and 2.0 ROMs and will **not** be fixed in the next ROM version.

   By the way, an interesting, but annoying, occurance of this bug happens when manually printing Legal sized documents with the 4.0 LaserWriter driver. When the Larger Print Area option in the style dialog is deselected (which is the default) the driver uses the `Note` operator to specify the page size. When the user prints the document using manual feed, and has a US letter tray in the printer, the image is shifted up on the page cutting off the top of the image. If you tell the user to turn on the Larger Print Area option in the style dialog, the driver specifies the page size using `Legal` instead of `Note` and the image is printed properly.

#124: Using Low-Level Printing Calls With AppleTalk ImageWriters

See also: The Printing Manager

Written by:	Ginger Jernigan	May 4, 1987
Update by:	Scott "ZZ" Zimmerman	Febuary ?, 1988
Updated:		March 1, 1988

When you use the low-level printer driver to print, you don't get the benefits of the error checking that is done when you use the high-level Printing Manager. So, if the user prints to an AppleTalk ImageWriter (including an AppleTalk ImageWriter LQ) that is busy printing another job, the driver doesn't know whether the printer is busy, offline, or disconnected. Because of this, PrError will return (and PrintErr will contain) `abortErr`.

Since there is no way to tell when you are printing to an AppleTalk ImageWriter, the only workaround for this is to use high-level Printing Manager interface.

#125: The Effect of Spool-a-page/Print-a-page on Shared Printers

See also: Printing Manager
Technical Note #72—
Optimizing for the LaserWriter—Techniques

Written by: Ginger Jernigan May 4, 1987
Updated: March 1, 1988

This technical note discusses drawbacks of using the spool-a-page/ print-a-page method of printing.

The "spool-a-page/print-a-page" method of printing prints each page of a document as a separate job instead of calling `PrPicFile` to print the entire picture file. Many applications adopted this method of printing to avoid running out of disk space while the ImageWriter driver was spooling the document to disk. As long as you are printing to a directly connected ImageWriter, you're fine, but if you are printing to remote or shared devices (like the AppleTalk ImageWriter and the LaserWriter), this method may create significant problems for the user.

When a job is initiated by the application, the driver establishes a connection with the printer via AppleTalk. When the job is completed, the driver closes the connection, allowing another job the opportunity to print. If each page is a job in itself, then the connection is closed and reopened between each page, allowing another application to print between the pages of the document, which, as you might imagine, could present a significant problem. If two people are printing to the same AppleTalk ImageWriter at the same time and their applications use the "spool-a-page/print-a-page" method of printing, the pages of each document will be interleaved at the printer.

Although there are good reasons for using this method of printing, it is only useful for a directly connected printer. From a compatibility point of view, this method of printing is built-in device dependence. Also, this method could create serious problems for other types of remote devices. Therefore, we are recommending that applications avoid using this method indiscriminately. You should check available disk space to see how much room you have before you print. If there isn't enough space for your entire document, then print as much as you can (to minimize the interleaving) before starting another job. Whenever possible, applications should use the print loop described on page II-155 in The Printing Manager chapter of *Inside Macintosh*.

Macintosh
Technical Notes



Developer Technical Support

#126: Sub(Launching) from a High-Level Language

Revised by: Rich Collyer & Mark Johnson
Written by: Rick Blair & Jim Friedlander

April 1989
May 1987

Note: Developer Technical Support takes the view that launching and sublaunching are features which are best **avoided** for compatibility (and other) reasons, but we want to make sure that when it is absolutely necessary to implement it, it is done in the safest possible way.

This Technical Note discusses the "safest" method of calling `_Launch` from a high-level language that supports inline assembly language with the option of launching or sublaunching another application.

Changes since August 1988: Incorporated Technical Note #52 on calling `_Launch` from a high-level language, changed the example to offer a choice between launching or sublaunching, added a discussion of the `_Launch` trap under MultiFinder, and updated the MPW C code to include inline assembly language.

The Segment Loader chapter of *Inside Macintosh* II-53 states the following about the `_Launch` trap:

> *"The routines below are provided for advanced programmers; they can be called only from assembly language."*

While this statement is technically true, it is easy to call `_Launch` from any high-level language which supports inline assembly code, and this Note provides examples of calling `_Launch` in MPW Pascal and C.

Before calling `_Launch`, you need to declare the inline procedure, which takes a variable of type `pLaunchStruct` as a parameter. Since the compiler pushes a pointer to this parameter on the stack, you need to include code to put this pointer into A0. The way to do this is with a `MOVE.L (SP)+,A0` instruction, which is `$205F` in hexadecimal, so the first word after `INLINE` is `$205F`. This instruction sets up `A0` to contain a pointer to the filename and `4(A0)` to contain the configuration parameter, so the last part of the inline is the `_Launch` trap itself, which is `$A9F2` in hexadecimal. The configuration parameter, which is normally zero, determines whether the application uses alternate screen and sound buffers. Since not all Macintosh models support these alternate buffers, you should avoid using them unless you have a specific circumstance which requires them.

The Finder does a lot of hidden cleanup and other tasks without user knowledge; therefore, it is best if you do not try to replace the Finder with a "mini" or try to launch other programs and have them return to your application. In the future, the Finder may provide better integration for applications, and you will circumvent this if you try to act in its place by sublaunching other programs.

If you have a situation where your application **must** launch another and have it return, and where you are not worried about incompatibility with future System Software versions, there is a "preferred" way of doing this which fits into the current system well. System file version 4.1 (or later) includes a mechanism for allowing a call to another application; we term this call a "sublaunch." You can perform a sublaunch by adding a set of simple extensions to the parameter block you pass to the `_Launch` trap.

_Launch and MultiFinder

Under MultiFinder, a sublaunch behaves differently than under the Finder. The application you sublaunch becomes the foreground application, and when the user quits that application, the system returns control to the next frontmost layer, which will not necessarily be your application.

If you set both high bits of `LaunchFlags`, which requests a sublaunch, your application will continue to execute after the call to `_Launch`. Under MultiFinder, the actual launch (and suspend of your application) will not happen in the `_Launch` trap, but rather after a call or more to `_WaitNextEvent`.

Under MultiFinder, `_Launch` currently returns an error if there is not enough memory to launch the desired application, if it cannot locate the desired application, or if the desired application is already open. In the latter case, that application will **not** be made active. If you attempted to launch, MultiFinder will call `_SysBeep`, your application will terminate, and control will given to the next frontmost layer. If you attempted to sublaunch, control will return to your application, and it is up to you to report the error to the user.

Currently, `_Launch` returns an error in register `D0` for a sublaunch, and you should check it for errors (`D0<0`) after any attempts at sublaunching. If `D0>=0` then your sublaunch was successful.

You should refer to the *Programmer's Guide to MultiFinder* (APDA) and Macintosh Technical Notes #180, MultiFinder Miscellanea and #205, MultiFinder Revisited: The 6.0 System Release, for further discussion of the `_Launch` trap under MultiFinder.)

Working Directories and Sublaunching With the Finder

Putting aside the compatibility issue for the moment, the only problem sublaunching creates under the **current** system is one of Working Directory Control Blocks (WDCBs). Unless the application you are launching is at the root directory or on an MFS volume, you must create a new WDCB and set it as the current directory when you launch the application.

In the example which follows, the new working directory is opened (allocated) by Standard File and its `WDRefNum` is returned in `reply.vRefNum`. If you do not use Standard File and cannot assume, for instance, that the application was in the blessed folder or root directory, then you must open a new working directory explicitly via a call to `_OpenWD`. You should give the new WDCB a `WDProcID` of `'ERIK'`, so the Finder (or another shell) would know to deallocate when it saw it was allocated by a "sublaunchee."

Although the sublaunching process is recursive (i.e., programs which are sublaunched may, in turn, sublaunch other programs), there is a limit of 40 on the number of WDCBs which can be created. With this limit, you could run out of available WDCBs very quickly if many programs were playing the shell game or neglecting to deallocate the WDCBs they had created. Make sure you check for **all** errors after calling `_PBOpenWD`. A `tMWDOErr` (–121) means that all available

WDCBs have been allocated, and if you receive this error, you should alert the user that the sublaunch failed and continue as appropriate.

> **Warning:** Although the example included in this Note covers sublaunching, Developer Technical Support strongly recommends that developers **not** use this feature of the _Launch trap. This trap will change in the not-too-distant future, and when it does change, applications which perform sublaunching **will** break. The only circumstance in which you could consider sublaunching is if you are implementing an integrated development system and are prepared to deal with the possibility of revising it every time Apple releases a new version of the System Software.

MPW Pascal

```
{It is assumed that the Signals are caught elsewhere; see Technical
 Note #88 for more information on the Signal mechanism}

{the extended parameter block to _Launch}
TYPE
   pLaunchStruct = ^LaunchStruct;
   LaunchStruct = RECORD
      pfName      : StringPtr;
      param       : INTEGER;
      LC          : PACKED ARRAY[0..1] OF CHAR; {extended parameters:}
      extBlockLen : LONGINT; {number of bytes in extension = 6}
      fFlags      : INTEGER; {Finder file info flags (see below)}
      launchFlags : LONGINT; {bit 31,30=1 for sublaunch, others reserved}
   END; {LaunchStruct}

FUNCTION LaunchIt(pLaunch: pLaunchStruct): OSErr; {< 0 means error}
   INLINE $205F, $A9F2, $3E80;
{ pops pointer into A0, calls Launch, pops D0 error code into result:
  MOVE.L  (A7)+,A0
  _Launch
  MOVE.W  D0,(A7)  ; since it MAY return }

PROCEDURE DoLaunch(subLaunch: BOOLEAN);  {Sublaunch if true and launch if false}

      VAR
         myLaunch     : LaunchStruct;   {launch structure}
         where        : Point;          {where to display dialog}
         reply        : SFReply;        {reply record}
         myFileTypes  : SFTypeList;     {we only want APPLs}
         numFileTypes : INTEGER;
         myPB         : CInfoPBRec;
         dirNameStr   : str255;

      BEGIN
         where.h := 20;
         where.v := 20;
         numFileTypes:= 1;
         myFileTypes[0]:= 'APPL';       {applications only!}
      {Let the user choose the file to Launch}
         SFGetFile(where, '', NIL, numFileTypes, myFileTypes, NIL, reply);
```

```
   IF reply.good THEN BEGIN
      dirNameStr:= reply.fName;   {initialize to file selected}

{Get the Finder flags}
      WITH myPB DO BEGIN
          ioNamePtr:= @dirNameStr;
          ioVRefNum:= reply.vRefNum;
          ioFDirIndex:= 0;
          ioDirID:= 0;
      END; {WITH}
      Signal(PBGetCatInfo(@MyPB,FALSE));
{Set the current volume to where the target application is}
      Signal(SetVol(NIL, reply.vRefNum));

{Set up the launch parameters}
      WITH myLaunch DO BEGIN
         pfName := @reply.fName;    {pointer to our fileName}
         param := 0;                {we don't want alternate screen or sound buffers}
         LC := 'LC';                {here to tell Launch that there is non-junk next}
         extBlockLen := 6;          {length of param. block past this long word}
         {copy flags; set bit 6 of low byte to 1 for RO access:}
         fFlags := myPB.ioFlFndrInfo.fdFlags;  {from GetCatInfo}

{Test subLaunch and set LaunchFlags accordingly}
      IF subLaunch THEN
         LaunchFlags := $C0000000              {set BOTH high bits for a sublaunch}
      ELSE
         LaunchFlags := $00000000;             {Just launch then quit}
      END; {WITH}

 {launch; you might want to put up a dialog which explains that
  the selected application couldn't be launched for some reason.}
      Signal(LaunchIt(@myLaunch));
   END; {IF reply.good}

END; {DoLaunch}
```

MPW C

```
typedef struct LaunchStruct {
        char            *pfName;                /* pointer to the name of launchee */
        short int       param;
        char            LC[2];                  /*extended parameters:*/
        long int        extBlockLen;            /*number of bytes in extension == 6*/
        short int       fFlags;                 /*Finder file info flags (see below)*/
        long int        launchFlags;            /*bit 31,30==1 for sublaunch, others reserved*/
} *pLaunchStruct;

pascal OSErr LaunchIt( pLaunchStruct pLnch) /* < 0 means error */
        = {0x205F, 0xA9F2, 0x3E80};

/* pops pointer into A0, calls Launch, pops D0 error code into result:
          MOVE.L  (A7)+,A0
          _Launch
          MOVE.W  D0,(A7)  ; since it MAY return */

OSErr DoLaunch(subLaunch)
        Boolean                 subLaunch;      /* Sublaunch if true and launch if false   */
{  /* DoLaunch */
        struct LaunchStruct     myLaunch;
        Point                   where;          /*where to display dialog*/
        SFReply                 reply;          /*reply record*/
        SFTypeList              myFileTypes;    /* we only want APPLs */
        short int               numFileTypes=1;
        HFileInfo               myPB;
        char                    *dirNameStr;
        OSErr                   err;

        where.h = 80;
        where.v = 90;
        myFileTypes[0] = 'APPL';                /* we only want APPLs */
        /*Let the user choose the file to Launch*/
        SFGetFile(where, "", nil, numFileTypes, myFileTypes, nil, &reply);

        if (reply.good)
        {
                dirNameStr = &reply.fName;      /*initialize to file selected*/

        /*Get the Finder flags*/
                myPB.ioNamePtr= dirNameStr;
                myPB.ioVRefNum= reply.vRefNum;
                myPB.ioFDirIndex= 0;
                myPB.ioDirID = 0;
                err = PBGetCatInfo((CInfoPBPtr) &myPB,false);
                if (err != noErr)
                        return err;

        /*Set the current volume to where the target application is*/
                err = SetVol(nil, reply.vRefNum);
                if (err != noErr)
                        return err;

        /*Set up the launch parameters*/
                myLaunch.pfName = &reply.fName;     /*pointer to our fileName*/
                myLaunch.param = 0;                 /*we don't want alternate screen
                                                      or sound buffers*/
        /*set up LC so as to tell Launch that there is non-junk next*/
                myLaunch.LC[0] = 'L'; myLaunch.LC[1] = 'C';
                myLaunch.extBlockLen = 6;           /*length of param. block past
                                                      this long word*/
        /*copy flags; set bit 6 of low byte to 1 for RO access:*/
                myLaunch.fFlags = myPB.ioFlFndrInfo.fdFlags;      /*from _GetCatInfo*/
```

```
        /* Test subLaunch and set launchFlags accordingly */
        if ( subLaunch )
                myLaunch.launchFlags = 0xC0000000;  /*set BOTH hi bits for a sublaunch  */
        else
                myLaunch.launchFlags = 0x00000000;  /* Just launch then quit           */

                err = LaunchIt(&myLaunch);          /* call _Launch                    */
                if (err < 0)
                {
                /* the launch failed, so put up an alert to inform the user */
                        LaunchFailed();
                        return err;
                }
                else
                        return noErr;
        } /*if reply.good*/
} /*DoLaunch*/
```

Further Reference:

- *Inside Macintosh*, Volumes I-12, II-53, & IV-83, The Segment Loader
- *Programmer's Guide to MultiFinder* (APDA)
- Technical Note #129, _SysEnvirons: System 6.0 and Beyond
- Technical Note #180, MultiFinder Miscellanea
- Technical Note #205, MultiFinder Revisited: The 6.0 System Release

#127: TextEdit EOL Ambiguity

See also: TextEdit

Written by: Rick Blair May 4, 1987
Updated: March 1, 1988

`TESetSelect` may be used to position the insertion point at the end of a line. There is an ambiguity, though; should the insertion point appear at the end of the preceding line or the start of the following one? It is possible to determine what will happen, as you are about to see.

There is an internal flag used by TextEdit to determine where the insertion point at the end of a line appears. This flag is part of the `clikStuff` field in the `TERec`. It is there mainly for the use of `TEClick`, but it is also used by `TESetSelect` (although it defaults to the right side of the previous line).

The following code can be used to force the insertion point to appear at the left of the following line when it is positioned at the end of a line; in MPW Pascal:

```
TEDeactivate(tH);
tH^^.clikStuff := 255;                          {position caret on left}
TESetSelect(eolcharpos, eolcharpos, tH);        {ambiguous point}
TEActivate(tH);
```

In MPW C:

```
TEDeactivate(tH);
(**tH).clikStuff = 255;                         /*position caret on left*/
TESetSelect(eolcharpos, eolcharpos, tH);        /*ambiguous point*/
TEActivate(tH);
```

If you want to ensure that the caret is on the right side (to which it normally defaults) then substitute a zero for the 255.

#128: PrGeneral

See also: The Printing Manager
Technical Note #118—
How to Check and Handle Printing Errors

Written by: Ginger Jernigan May 4, 1987
Updated: March 1, 1988

The Printing Manager architecture has been expanded to include a new procedure called `PrGeneral`. The features described here are advanced, special-purpose features, intended to solve specific problems for those applications that need them. The calls to determine printer resolution introduce a good deal of complexity into the application's code, and should be used only when necessary.

Version 2.5 (and later) of the ImageWriter driver and version 4.0 (and later) of the LaserWriter driver implement a generic Printing Manager procedure called `PrGeneral`. This procedure allows the Print Manager to expand in functionality, by allowing printer drivers to implement various new functions. The Pascal declaration of `PrGeneral` is:

```
PROCEDURE PrGeneral (pData: Ptr);
```

The `pData` parameter is a pointer to a data block. The structure of the data block is declared as follows:

```
TGnlData = RECORD {1st 8 bytes are common for all PrGeneral calls)
   iOpCode    : INTEGER;  {input}
   iError     : INTEGER;  {output}
   lReserved : LONGINT;  {reserved for future use}
   {more fields here, depending on particular call}
END;
```

The first field is a 2-byte opcode, `iOpCode`, which acts like a routine selector. The currently available opcodes are described below.

The second field is the error result, `iError`, which is returned by the print code. This error only reflects error conditions that occur during the `PrGeneral` call. For example, if you use an opcode that isn't implemented in a particular printer driver then you will get a `OpNotImpl` error.

Here are the errors currently defined:

```
CONST
   noErr = 0;                    {everything's hunky}
   NoSuchRsl = 1;                {the resolution you chose isn't available}
   OpNotImpl = 2;                {the driver doesn't support this opcode}
```

After calling PrGeneral you should always check PrError. If noErr is returned, then you can proceed. If ResNotFound is returned, then the current printer driver doesn't support PrGeneral and you should proceed appropriately. See Technical Note #118 for details on checking errors returned by the Printing Manager.

IError is followed by a four byte reserved field (that means don't use it). The contents of the rest of the data block depends on the opcode that the application uses. There are currently five opcodes used by the ImageWriter and LaserWriter drivers.

The Opcodes

Initially, the following calls are implemented via PrGeneral:

- GetRslData (get resolution data): iOpCode = 4
- SetRsl (set resolution): iOpCode = 5
- DraftBits (bitmaps in draft mode): iOpCode = 6
- noDraftBits (no bitmaps in draft mode): iOpCode = 7
- GetRotn (get rotation): iOpCode = 8

The GetRslData and SetRsl allow the application to find out what physical resolutions the printer supports, and then specify a supported resolution. DraftBits and noDraftBits invoke a new feature of the ImageWriter, allowing bitmaps (imaged via CopyBits) to be printed in draft mode. GetRotn lets an application know whether landscape has been selected. Below is a detailed description of how each routine works.

The GetRslData Call

GetRslData (iOpCode = 4) returns a record that lets the application know what resolutions are supported by the current printer. The application can then use SetRsl (description follows) to tell the printer driver which one it will use. This is the format of the input data block for the GetRslData call:

```
TRslRg = RECORD              {used in TGetRslBlk}
   iMin, iMax: Integer;      {0 if printer only supports discrete resolutions}
END;

TRslRec = RECORD             {used in TGetRslBlk}
   iXRsl, iYRsl: Integer;    {a discrete, physical resolution}
END;
```

```
TGetRslBlk = RECORD      {data block for GetRslData call}
   iOpCode:    Integer;  {input; = getRslDataOp}
   iError:     Integer;  {output}
   lReserved:  LongInt;  {reserved for future use}
   iRgType:    Integer;  {output; version number}
   XRslRg:     TRslRg;   {output; range of X resolutions}
   YRslRg:     TRslRg;   {output; range of Y resolutions}
   iRslRecCnt: Integer;  {output; how many RslRecs follow}
   rgRslRec:   ARRAY[1..27] OF TRslRec;  {output; number filled depends on
                                          printer type}
END;
```

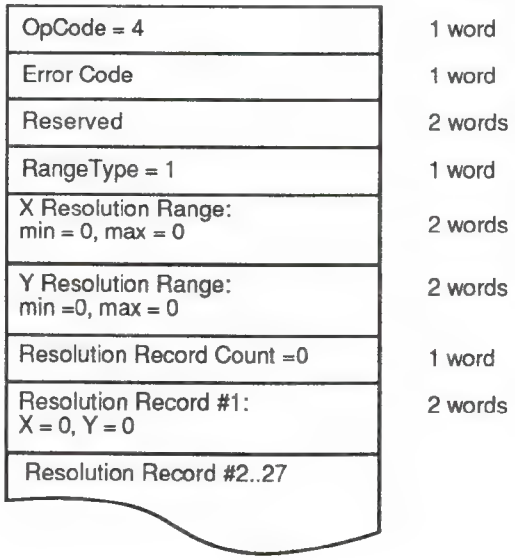
The `iRgType` field is much like a version number; it determines the interpretation of the data that follows. At present, a `iRgType` value of 1 applies both to the LaserWriter and to the ImageWriter.

For variable-resolution printers like the LaserWriter, the resolution range fields `XRslRg` and `YRslRg` express the ranges of values to which the X and Y resolutions can be set. For discrete-resolution printers like the ImageWriter, the values in the resolution range fields are zero.

Note: In general, X and Y in these records are the horizontal and vertical directions of the **printer**, not the document! In landscape orientation, X is horizontal on the printer but vertical on the document.

After the resolution range information there is a word which gives the number of resolution records that contain information. These records indicate the physical resolutions at which the printer can actually print dots. Each resolution record gives an X value and a Y value.

When you call `PrGeneral` you pass in a data block that looks like this:



Below is the data block returned for the LaserWriter:

OpCode = 4	1 word
Error Code (0 = okay)	1 word
Reserved	2 words
RangeType = 1	1 word
X Resolution Range: min = 72, max = 1500	2 words
Y Resolution Range: min = 72, max = 1500	2 words
Resolution Record Count = 1	1 word
Resolution Record #1: X = 300, Y = 300	2 words

Note that all the resolution range numbers happen to be the same for this printer. There is only one resolution record, which gives the physical X and Y resolutions of the printer (300x300).

Below is the data block returned for the ImageWriter.

OpCode = 4	1 word
Error Code (0 = okay)	1 word
Reserved	2 words
RangeType = 1	1 word
X Resolution Range: min =0, max = 0	2 words
Y Resolution Range: min = 0, max = 0	2 words
Resolution Record Count = 4	1 word
Resolution Record #1: X = 72, Y = 72	2 words
Resolution Record #2: X =144, Y = 144	2 words
Resolution Record #3: X = 80, Y = 72	2 words
Resolution Record #4: X = 160, Y = 144	2 words

All the resolution range values are zero, because only discrete resolutions can be specified for this printer. There are four resolution records giving these discrete physical resolutions.

Note that `GetRslData` always returns the same information for a particular printer type—it is **not** dependent on what the user does or on printer configuration information.

The SetRsl Call

SetRsl (iOpCode = 5) is used to specify the desired imaging resolution, after using GetRslData to determine a workable pair of values. Below is the format of the data block:

```
TSetRslBlk =   RECORD     {data block for SetRsl call}
   iOpCode:    Integer;   {input; = setRslOp}
   iError:     Integer;   {output}
   lReserved:  LongInt;   {reserved for future use}
   hPrint:     THPrint;   {input; handle to a valid print record}
   iXRsl:      Integer;   {input; desired X resolution}
   iYRsl:      Integer;   {input; desired Y resolution}
END;
```

hPrint should be the handle of a print record that has previously been passed to PrValidate. If the call executes successfully, the print record is updated with the new resolution; the data block comes back with 0 for the error and is otherwise unchanged.

However, if the desired resolution is not supported, the error is set to noSuchRsl and the resolution fields are set to the printer's default resolution

Note that you can undo the effect of a previous call to SetRsl by making another call that specifies an unsupported resolution (such as 0x0), forcing the default resolution.

The DraftBits Call

DraftBits (iOpCode = 6) is implemented on both the ImageWriter and the LaserWriter. (On the LaserWriter it does nothing, since the LaserWriter is always in draft mode and can always print bitmaps.) Below is the format of the data block:

```
TDftBitsBlk =  RECORD     {data block for DraftBits and NoDraftBits calls}
   iOpCode:    Integer;   {input; = draftBitsOp or noDraftBitsOp}
   iError:     Integer;   {output}
   lReserved:  LongInt;   {reserved for future use}
   hPrint:     THPrint;   {input; handle to a valid print record}
END;
```

hPrint should be the handle of a print record that has previously been passed to PrValidate.

This call forces draft-mode (i.e., immediate) printing, and will allow bitmaps to be printed via CopyBits calls. The virtue of this is that you avoid spooling large masses of bitmap data onto the disk, and you also get better performance.

The following restrictions apply:

- This call should be made before bringing up the print dialogs because it affects their appearance. On the ImageWriter, calling DraftBits disables the landscape icon in the Style dialog, and the Best, Faster, and Draft buttons in the Job dialog.

- If the printer does not support draft mode, already prints bitmaps in draft mode, or does not print bitmaps at all, this call does nothing.
- Only text and bitmaps can be printed.
- As in the normal draft mode, landscape format is not allowed.
- Everything on the page must be strictly Y-sorted, i.e. no reverse paper motion between one string or bitmap and the next. Note that this means you can't have two or more objects (text or bitmaps) side by side; the top boundary of each object must be no higher than the bottom of the preceding object.

The last restriction is important. If you violate it, you will not like the results. But note that if you want two or more bitmaps side by side, you can combine them into one before calling `CopyBits` to print the result. Similarly, if you are just printing bitmaps you can rotate them yourself to achieve landscape printing.

The NoDraftBits Call

`NoDraftBits` (`iOpCode` = 7) is implemented on both the ImageWriter and the LaserWriter. (On the LaserWriter it does nothing, since the LaserWriter is always in draft mode and can always print bitmaps.) The format of the data block is the same as that for the `DraftBits` call.

This call cancels the effect of any preceding `DraftBits` call. If there was no preceding `DraftBits` call, or the printer does not support draft-mode printing anyway, this call does nothing.

The GetRotn Call

`GetRotn` (`iOpCode` = 8) is implemented on the ImageWriter and LaserWriter. Here is the format of the data block:

```
TGetRotnBlk =  RECORD          {data block for GetRotn call}
   iOpCode:    Integer;        {input; = getRotnOp}
   iError:     Integer;        {output}
   lReserved:  LongInt;        {reserved for future use}
   hPrint:     THPrint;        {input; handle to a valid print record}
   fLandscape: Boolean;        {output; Boolean flag}
   bXtra:      SignedByte;     {reserved}
END;
```

`hPrint` should be the handle to a print record that has previously been passed to `PrValidate`.

If landscape orientation is selected in the print record, then `fLandscape` is true.

How To Use The PrGeneral Opcodes

The `SetRsl` and `DraftBits` calls may require the print code to suppress certain options in the Style and/or Job dialogs, therefore they should always be called before any call to the Style or Job dialogs. An application might use these calls as follows:

- Get a new print record by calling `PrintDefault`, or take an existing one from a document and call `PrValidate` on it.
- Call `GetRslData` to find out what the printer is capable of, and decide what resolution to use. Check `PrError` to be sure the `PrGeneral` call is supported on this version of the print code; if the error is `ResNotFound`, you have older print code and must print accordingly. But if the PrError return is 0, proceed:
- Call `SetRsl` with the print record and the desired resolution if you wish.
- Call `DraftBits` to invoke the printing of bitmaps in draft mode if you wish.

Note that if you call either `SetRsl` or `DraftBits`, you should do so before the user sees either of the printing dialogs.

Macintosh
Technical Notes



Developer Technical Support

#129: _Gestalt & _SysEnvirons—a Never-Ending Story

Revised by: Dave Radcliffe May 1992
Written by: Jim Friedlander May 1987

This Technical Note discusses the latest changes and enhancements in the `_Gestalt` and `_SysEnvirons` calls.

Changes since October 1991: Clarified information on Gestalt information for Macintosh PowerBook computers and added information on the Macintosh LC II and the `gestaltHardwareAttr` selector.

Introduction

Previous versions of this Note provided the latest documentation on new information the `_SysEnvirons` trap could return. DTS will continue to revise this Note to provide this information; however, as the `_Gestalt` trap is now the preferred method for determining information about a machine environment, this Note will also provide up-to-date information on `_Gestalt` selectors.

_Gestalt

This Note now documents `_Gestalt` selectors and return values added since the release of *Inside Macintosh* Volume VI. Please note that this is supplemental information; for the complete description of `_Gestalt` and its use, please refer to *Inside Macintosh* Volume VI.

The Macintosh LC II is identical to the Macintosh LC, except for the presence of an MC68030 processor, so it returns the *same* gestaltMachineType response as the Macintosh LC (i.e. 19). Developers are reminded that the gestaltMachineType selector is for informational purposes only and should not be used as a basis for programmatic decisions. As always, developers are encouraged to test for the specific features they need and not to rely on any particular machine having a particular set of features.

Note: The *Macintosh PowerBook 100 Developer Notes* and the *Macintosh PowerBook 140/170 Developer Notes*, available from APDA and on the *Developer CD Series* disc and AppleLink, incorrectly document `gestaltMachineType` response values for the Macintosh PowerBook computers. The following values are, and have always been, the correct values.

Additional Gestalt Response Values

```
{ gestaltMachineType response values }
        gestaltQuadra900                = 20;     { Macintosh Quadra 900 }
        gestaltPowerBook170             = 21;     { Macintosh PowerBook 170 }
        gestaltQuadra700                = 22;     { Macintosh Quadra 700 }
        gestaltClassicII                = 23;     { Macintosh Classic II }
        gestaltPowerBook100             = 24;     { Macintosh PowerBook 100 }
        gestaltPowerBook140             = 25;     { Macintosh PowerBook 140 }

{ gestaltKeyboardType response values }
        gestaltPwrBookADBKbd            = 12;     { PowerBook Keyboard }
        gestaltPwrBookISOADBKbd         = 13;     { PowerBook Keyboard (ISO) }
```

gestaltHardwareAttr Selector

The `gestaltHardwareAttr` selector has been a source of confusion for developers since originally documented in *Inside Macintosh* Volume VI . This section will try to reduce that confusion and also introduce additional information returned by the selector. But be warned that use of this selector for anything other than informational purposes should be deemed a compatibility risk. In other words, if you are dependent on the information returned by this selector to function on existing computers, you will almost certainly have problems on future systems.

The reason for this is that `gestaltHardwareAttr` returns very low-level hardware information. If you need to use this information, it implies you are too hardware dependent. So be very careful about using this information.

The principal source of confusion is bit 7, described as `gestaltHasSCSI`. What this bit really means is the machine is equipped with SCSI based on the 53C80 chip, which was introduced in the Macintosh Plus. This bit will be zero on the Macintosh IIfx and the Macintosh Quadra computers because they have a different low-level SCSI implementation. The Macintosh IIfx has a 53C80 compatible chip that also supports SCSI DMA. It reports this information using bit 6 of the `gestaltHardwareAttr` response. The Macintosh Quadra computers have yet another SCSI implementation based on the 53C96 chip and so report different information (see below).

Another source of confusion is bit 4 (`gestaltHasSCC`). The Macintosh IIfx and Macintosh Quadra 900 have intelligent I/O processors (IOPs) that normally isolate the hardware and make direct access to the SCC impossible. Normally, these machines will report that they do not have an SCC implying, correctly, that were you to attempt to access it directly, you would fail. However, if the user has used the Compatibility Switch control panel to enable compatibility mode, `gestaltHasSCC` will report true indicating you may access the SCC directly. But remember that doing so means you are doing direct hardware access and that there may be a day when you can't access the SCC under any circumstances.

New gestaltHardwareAttr Values for Macintosh Quadra Computers

Below are the new bits supported by the Macintosh Quadra computers. Any other bits remain undocumented and subject to change.

```
        gestaltHasSCSI961                   = 21; { 53C96 SCSI controller on internal bus
}
```

```
        gestaltHasSCSI962                   = 22; { 53C96 SCSI controller on external bus
}
```

_SysEnvirons

_SysEnvirons was the standard way to determine the features available on a given machine. The preferred method to get this information is now _Gestalt; information on _SysEnvirons is now provided only for backward compatibility.

As originally conceived, _SysEnvirons would check the versionRequested parameter to determine what level of information you were prepared to handle, but this technique means updating _SysEnvirons for every new hardware product Apple produces. With system software version 6.0, _SysEnvirons introduced version 2 of environsVersion to provide information about new hardware as we introduce it; this new version returns the same SysEnvRec as version 1.

Beginning with system software version 6.0.1, Apple releases a new version of _SysEnvirons only when engineering makes changes to its structure (that is, when they add new fields to SysEnvRec); all existing versions return accurate information about the machine environment even if part of that information was not originally defined for the version you request. For example, if you call _SysEnvirons with versionRequested = 1 on a Macintosh IIfx, it returns a machineType of envMacIIfx even though this machine type originally was not defined for version 1 of the call.

You should use version 2 of _SysEnvirons until Apple releases a newer version. MPW 3.0 defines a constant curSysEnvVers, which can be used to minimize the need for source code revisions when _SysEnvirons evolves. Regardless of the version used, however, your software should be prepared to handle unexpected values and should not make assumptions about functionality based on current expectations. For example, if your software currently requires a Macintosh II, testing for machineType >= envMacII may result in your software trying to run on a machine that does not support the features it requires, so test for specific functionality (that is, hasFPU, hasColorQD, and so on).

Warning: This test for specific functionality is particularly true of FPUs (floating-point units). Some CPUs, such as the Macintosh IIsi, may have optional, user-installed FPUs; therefore, an application should not assume that any Macintosh with a microprocessor greater than a 68000 (for example, 68020, 68030 or 68040) has an FPU (68881/68882 or built-in for the 68040). If an application makes a conditional branch to execute floating-point instructions directly, then it should first explicitly check for the presence of the FPU.

You should always check the environsVersion when returning from _SysEnvirons since the glue always returns as much information as possible, with environsVersion indicating the highest version available, even if the call returns an envSelTooBig (–5502) error.

Calling _SysEnvirons From a High-Level Language

Due to a documentation error in *Inside Macintosh* Volume V, DTS still receives questions about how to call _SysEnvirons properly from Pascal and C. *Inside Macintosh* defines the Pascal interface to _SysEnvirons as follows:

```
FUNCTION SysEnvirons (versRequested: INTEGER; VAR theWorld: SysEnvRecPtr) : OSErr;
```

Because `theWorld` is passed by reference (as a `VAR` parameter), it is not correct to pass a `SysEnvRecPtr` in the second argument. Pascal would then generate a pointer to this pointer and pass that to the `_SysEnvirons` trap in `A0`. (The assembly-language information is essentially correct; `_SysEnvirons` really does want a pointer to a `SysEnvRec` in `A0`.) The correct Pascal interface to `_SysEnvirons` is therefore:

```
FUNCTION SysEnvirons (versionRequested: INTEGER; VAR theWorld: SysEnvRec) : OSErr;
```

In this case, Pascal pushes a pointer to `theWorld` on the stack. The Pascal interface glue then pops this pointer off the stack directly into `A0` and calls `_SysEnvirons`. Everything is copacetic.

C programmers should recognize their corresponding interface:

```
pascal OSErr SysEnvirons (short versionRequested, SysEnvRec *theWorld);
```

Inside Macintosh defines the type `SysEnvPtr = ^SysEnvRec`. It also sometimes refers to this type as `SysEnvRecPtr`. The inconsistency is insignificant because in reality MPW does not define any such type, under either name; therefore, it is never needed.

Inside Macintosh also states that "all of the Toolbox Managers must be initialized before calling SysEnvirons." This statement is not necessarily true. Startup documents (INITs), for instance, may wish to call `_SysEnvirons` without initializing any of the Toolbox Managers. Keep in mind that the `atDrvrVersNum` field returns a zero result if the AppleTalk drivers are not initialized. The system version, machine type, processor type, and other key data return normally.

Additional _SysEnvirons Constants

The following are new `_SysEnvirons` constants which are not documented in *Inside Macintosh*; however, you should refer to *Inside Macintosh* Volume V-1, Compatibility Guidelines, for the rest of the story.

machineType

```
envMacIIx               = 5;    { Macintosh IIx }
envMacIIcx              = 6;    { Macintosh IIcx }
envSE30                 = 7;    { Macintosh SE/30 }
envPortable             = 8;    { Macintosh Portable }
envMacIIci              = 9;    { Macintosh IIci }
envMacIIfx              = 11;   { Macintosh IIfx }
envMacClassic           = 15;   { Macintosh Classic }
envMacIIsi              = 16;   { Macintosh IIsi }
envMacLC                = 17;   { Macintosh LC }
envMacQuadra900         = 18;   { Macintosh Quadra 900 }
envMacPowerBook170      = 19;   { Macintosh PowerBook 170 }
envMacQuadra700         = 20;   { Macintosh Quadra 700 }
envMacClassicII         = 21;   { Macintosh Classic II }
envMacPowerBook100      = 22;   { Macintosh PowerBook 100 }
envMacPowerBook140      = 23;   { Macintosh PowerBook 140 }
```

processor

```
env68030                = 4;    { MC68030 processor }
env68040                = 5;    { MC68040 processor }
```

keyBoardType

```
envPrtblADBKbd          = 6;    { Portable Keyboard }
envPrtblISOKbd          = 7;    { Portable Keyboard (ISO) }
envStdISOADBKbd         = 8;    { Apple Standard Keyboard (ISO) }
envExtISOADBKbd         = 9;    { Apple Extended Keyboard (ISO)

envADBKbdII             = 10;   { Apple Keyboard II }
envADBISOKbdII          = 11;   { Apple Keyboard II (ISO) }
envPwrBkADBKbd          = 12;   { PowerBook Keyboard }
envPwrBkISOKbd          = 13;   { PowerBook Keyboard (ISO) }
```

Further Reference:

- *Inside Macintosh*, Volumes V and VI, Compatibility Guidelines

#130: Clearing ioCompletion

See also: The File Manager

Written by: Jim Friedlander
Updated:

May 4, 1987
March 1, 1988

When making **s**ynchronous calls to the File Manager, it is not necessary to clear `ioCompletion` field of the parameter **b**lock, since that is done for you.

Some earlier technotes explicitly cleared `ioCompletion`, with the knowledge that this was unnecessary, to try to encourage developers to fill in all fields of parameter blocks as indicated in *Inside Macintosh*.

By the way, this is true of all parameter calls—you only have to set fields that are explicitly required.

#131: TextEdit Bugs in System 4.2

Written by: Chris Derossi June 1, 1987
Updated: March 1, 1988

This note formerly described the known bugs with the version of Styled TextEdit that was provided with System 4.1. Many of these bugs were fixed in System 4.2. This updated Technical Note describes the remaining known problems.

TEStylInsert

Calling `TEStylInsert` while the TextEdit record is deactivated causes unpredictable results, so make sure to only call `TEStylInsert` when the TextEdit record is active.

TESetStyle

When using the `doFace` mode with `TESetStyle`, the style that you pass as a parameter is `ORed` into the style of the currently selected text. If you pass the empty set (no styles) though, `TESetStyle` is supposed to remove all styles from the selected text. But `TESetStyle` checks an entire word instead of just the high-order byte of the `tsFace` field. The style information is contained completely in the high-order byte, and the low-order byte may contain garbage.

If the low-order byte isn't zero, `TESetStyle` thinks that the `tsFace` field isn't empty, so it goes ahead and ORs it with the selected text's style. Since the actual style portion of the `tsFace` field is zero, no change occurs with the text. If you want to have `TESetStyle` remove all styles from the text, you can explicitly set the `tsFace` field to zero like this:

```
VAR
   myStyle  : TextStyle;
   anIntPtr : ^Integer;

BEGIN
   ...
   anIntPtr := @myStyle.tsFace;
   anIntPtr^ := 0;
   TESetStyle(doFace, myStyle, TRUE, textH);
   ...
END;
```

TEStylNew

The line heights array does not get initialized when `TEStylNew` is called. Because of this, the caret is initially drawn in a random height. This is easily solved by calling `TECalText` immediately after calling `TEStylNew`. Extra calls to `TECalText` don't hurt anything anyway, so this will be compatible with future Systems.

An extra character run is placed at the beginning of the text which corresponds to the font, size, and style which were in the grafPort when `TEStylNew` was called. This can cause the line height for the first line to be too large. To avoid this, call `TextSize` with the desired text size before calling `TEStylNew`. If the text's style information cannot be determined in advance, then call `TextSize` with a small value (like 9) before calling `TEStylNew`.

TEScroll

The bug documented in Technical Note #22 remains in the new TextEdit. `TEScroll` called with zero for both vertical and horizontal displacements causes the insertion point to disappear. The workaround is the same as before; check to make sure that `dV` and `dH` are not both zero before calling `TEScroll`.

Growing TextEdit Record

TextEdit is supposed to dynamically grow and shrink the `LineStarts` array in the `TERec` so that it has one entry per line. Instead, when lines are added, TextEdit expands the array without first checking to see if it's already big enough. In addition, TextEdit never reduces the size of this array.

Because of this, the longer a particular TextEdit record is used, the larger it will get. This can be particularly nasty in programs that use a single `TERec` for many operations during the program's execution.

Restoring Saved TextEdit Records

Applications have used a technique for saving and restoring styled text which involves saving the contents of all of the TextEdit record handles. When restoring, `TEStylNew` is called and the TextEdit record's handles are disposed. The saved handles are then loaded and put into the TextEdit record. This technique should not be used for the `nullStyle` handle in the style record.

Instead, when `TEStylNew` is called, the `nullStyle` handle from the style record should be copied into the saved style record. This will ensure that the fields in the null-style record point to valid data.

#132: AppleTalk Interface Update

See also: The AppleTalk Manager
Inside AppleTalk (for ZIP information)
Technical Note #121—
Using the High-Level AppleTalk Routines

Written by: Bryan Stearns July 1, 1987
Updated: March 1, 1988

Technical Note #121 announced that we would be moving to a simplified AppleTalk Manager interface. That interface is available now, as part of MPW 2.0 and newer.

Documentation for this new interface is contained in the AppleTalk Manager chapter of *Inside Macintosh Volume V*. This technical note contains some of the preliminary documentation for this interface and some useful points about information about it, and AppleTalk in general.

The original AppleTalk Pascal Interfaces, known as `ABPasIntf`, were designed to simplify use of AppleTalk from high-level languages. Instead, they've caused us a few compatibility problems. We've decided to encourage use of the same interface that assembly-language AppleTalk uses, a parameter-block interface in the same style as the low-level interfaces to the File and Device Managers.

The original calls are still supported (and will be for a while) as an "alternate" interface, but we suggest that you consider moving to the new "preferred" calls. Be warned that use of the original calls may cause compatibility problems with future system software. Also, new protocols (like ASP, the AppleTalk Session Protocol) are only provided with the new interfaces.

The new interface uses parameter blocks like those used by the File and Device Managers; you fill out the call-specific fields of the block, and a small amount of glue code (provided with development environments like MPW) turns the parameter block into a `Control` call to the appropriate AppleTalk driver.

Most calls have an interface like:

```
FUNCTION PSomeCall(thePBPtr: ATPPBptr; asyncFlag: BOOLEAN): OSErr;
```

The glue fills in the fields `csCode` and `ioRefNum` with the appropriate value for the call you're making.

Synchronous and Asynchronous calls

You can still make calls synchronously ("do it now") or asynchronously ("start it now, finish it soon"). If you choose to make a call asynchronously, be sure to provide a completion routine in the `ioCompletion` field (to be called when the call finally finishes), or poll the `ioResult` field of the parameter block (the call is done if `ioResult` is less than or equal to 0).

You must not move or dispose of a parameter block before the call finishes; when the call does complete, you are responsible for throwing the parameter block away (if you allocated it using Memory Manager routines).

Note that the alternate interfaces generated a network event on completion of an asynchronous call; this service is not provided by the preferred interfaces, partly because of future compatibility problems. See Technical Note #142 for background information.

Packed data structures

Several of the data structures used by the new interfaces are packed; Pascal doesn't deal well with these structures. Special calls are provided for building LAP and DPP write-data structures, NBP names-table elements, and ATP buffer data structures.

For example, when registering a name (using `PRegisterName`), you'll use a `NamesTableEntry` structure. This structure consists of a few unpacked fields, followed by an entity-name: three strings (representing the object, type, and zone fields of the name) packed together. You can call `NBPSetNTE` to pack the strings into the `NamesTableEntry` structure. When you remove the name (`PRemoveName`), you'll use the entity-name by itself; you can use `NBPSetEntity` to pack it in.

Zone Interface Protocol

A function, `GetBridgeAddress`, is provided to obtain the node ID of a bridge, for use in ZIP transactions (zero is returned if no bridge is present on your network). You make ZIP calls using ATP requests, as described in the *Inside AppleTalk* chapter on ZIP.

#133: Am I Talking To A Spooler?

See also: *PostScript Language Reference Manual*
Adobe Systems Document Structuring Conventions

Written by: Ginger Jernigan
Updated:

July 1, 1987
March 1, 1988

When the LaserShare spooler is on an AppleTalk network, it acts like a LaserWriter-type device, which can be chosen and communicated with much like a real LaserWriter. Some applications, however, must communicate with a LaserWriter directly, not a spooler. If this is true for your application, you can check whether you are actually talking to a real LaserWriter by sending to the LaserWriter the following query:

```
%!PS-Adobe-1.2 Query
%%Title: Query to Spooler/Non-Spooler status
%%?BeginSpoolerQuery
(0) = flush
%%?EndSpoolerQuery 1
%%EOF
```

(The query has to be sent using the Printer Access Protocol (PAP). The object code for PAP is available from Licensing.) If the string returned begins with a '%%' then it is a status string and you can ignore it and wait for another string. If the LaserWriter is actually a LaserShare spooler, then the string that is returned will be '1'. If the LaserWriter is a real LaserWriter then the string returned will be '0'.

#134: Hard Disk Medic & Booting Camp

See also: Hard Disk Users Manual
Technical Note 154—Macintosh Plus ROMs
Technical Note 113—Boot Blocks
Technical Note 67—Finding the 'Blessed Folder'

Written by: Bo3b Johnson
Updated:

July 1, 1987
March 1, 1988

The death of a hard disk with megabytes worth of data can be exceedingly traumatic. This technical note will describe techniques for recovering a hard disk and the data that is on it. The discussion will also include some tips on how to avoid problems.

You should never need this information. However, software problems can wreak havoc upon otherwise functional disks. When they have the equivalent of a heart attack, there are a number of steps that can be taken to try to recover the disk. There are occasions when the disk itself is not bad, and it may be possible to correct the disk without having to reformat the disk and restore the data from a backup. This note will describe some of the steps that can be used with Apple Hard Disks, but most of the information pertains to all hard disks. For example, the HD SC Setup program is specific to the Apple drives, but there is probably a similar utility for every hard disk. This is primarily a discussion of what to do from the user standpoint, but there are a few suggestions on ways of retrieving data via programmatic means.

This discussion will focus on the SCSI disks since they are more complex in terms of the booting sequence. For other hard disks, like the standard HD-20, most of the information still applies, but SCSI-specific sequences can be ignored. For example, the standard HD-20 also has an installer program, although it is different than HD SC Setup.

Attack of the Nasties

There are a number of unusual conditions that a hard disk may get itself in:

1) The data is intact, but the hard disk won't boot.
2) The SCSI disk won't boot and only shows up after running HD SC Setup.
3) The disk will boot but hangs part way through the boot process.
4) There are data errors while the disk is running.
5) The disk is very slow returning to the Finder.
6) The computer crashes or hangs when returning to the Finder.

7) The disk appears in a "This disk is bad" dialog.
8) The disk never shows up at all.

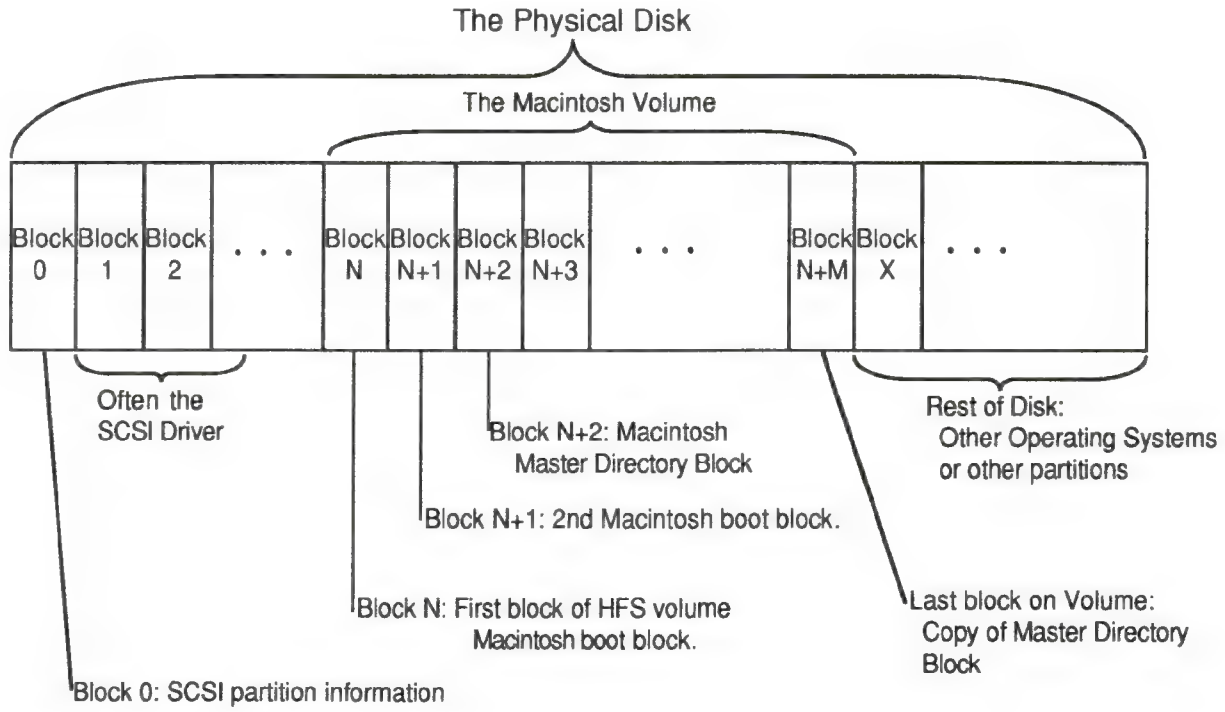
These problems can develop from a number of sources, including system crashes, rebooting at bad times, power fluctuations, malicious software, old software, buggy software, etc. In general, these problems will be software-related, since the hardware itself is very rarely defective.

This technical note will discuss:

1) The normal stages in the booting process.
2) Results of errors during the various stages in the booting process.
3) A step-by-step procedure to follow in order to maximize your chances of recovering the disk and the data.

A Boot to the Head

This discussion will detail a normal boot process of a Macintosh with a single hard disk attached. For clarity, this section will deliberately ignore potential problems and the complexities involved in different configurations. The following sections will detail some errors that may occur, and give more information in terms of what the ROM will do to boot the system. A SCSI disk can be thought of in the following fashion:



The important thing to note from this diagram is that the Macintosh volume is a subset of the entire SCSI Disk. There can be more than one Macintosh volume on a given disk, or even other volumes that are not Macintosh volumes.

1) Check the SCSI port:

Immediately after the RAM check, the system looks at the SCSI port to see if there are any drives connected. If a SCSI drive is found the system reads the SCSI partition information in block 0. This block is specific to SCSI drives and is always found at block 0 of the disk. The SCSI Manager then reads in the SCSI driver from the disk. Once the driver is loaded into memory, the system will use the driver to read and write blocks from the disk, instead of the ROM boot code. The driver reads and writes blocks relative to the beginning of the Macintosh volume on the SCSI drive, which can start anywhere on the physical disk.

2) Decide which disk is to be the startup disk:

The Macintosh then looks at the floppy disks to see if there is a disk that it should try to use. If so, it will always boot from the floppy. If there are no floppy disks, the startup hard disk is chosen. The Macintosh boot blocks are read off of the chosen disk to determine if the volume is bootable. The two Macintosh boot blocks (same boot blocks as those found on floppies) are read using the SCSI Driver. The Macintosh boot blocks are found as the first two blocks on the Macintosh **volume**, but are much higher in terms of where they are found on the **disk** itself. See the figure for the difference between the Macintosh volume and the SCSI disk. The driver cannot normally read the SCSI partition information, or any blocks outside of the Macintosh volume.

3) Execute the Macintosh boot blocks:

The boot blocks are composed of strings and parameters which determine various system functions, and code that finishes the job of booting the system.

The hard disk is mounted as a volume, using the `PBMountVol` call. The volume has the two Macintosh boot blocks, as well as the volume header. The `PBMountVol` will use the driver to read the volume header and other information from the disk. Once the volume is mounted, there are only volume reads and writes, and the driver is responsible for the actual SCSI disk reads.

The System file is opened on the volume. The patch code for the current ROM is read into the system, including the patches to the SCSI Manager.

The Finder is launched.

4) The Finder uses the Desktop file on the volume to draw the desktop.

The Icons that make up the desktop representation of the Macintosh volume are stored in the Desktop file. The Desktop file is invisible and used only by the Finder.

That is a rather simplistic view of the boot process. There are a number of complications that arise due to the wild variety of devices that can be attached to a Macintosh. The full boot process is essentially a series of special cases, leading to the final booted System at the Finder's desktop (or in the startup application). The following section will go into painstaking detail in order to give you enough information to determine what step in the boot process failed.

Tough Boots

To further explain the boot process:

1) Check the SCSI port:

a) Before starting the boot process, the screen will be filled with a grey pattern.
b) Before the Macintosh will check for any SCSI devices, it will first reset the SCSI bus using a `SCSIReset`. This is to make sure the bus was not left in a bad state.
c) The Macintosh will then start a cycle through all 7 SCSI IDs (from 6..0) to see which disks are connected, and keeps a table of all disks that are connected.
d) For each disk that is connected to the Macintosh, the ROM boot code will use the SCSI Manager to read in the SCSI partition information to find where the driver is located on the disk. The signature of the SCSI partition information is also checked to be sure that the device is valid.
e) The SCSI Manager will then be used to read the driver into memory. Once the driver is loaded for a given disk, the driver is called to install itself. The driver will usually post a Disk Inserted event to have its volume mounted by the Finder.
f) Steps d and e are repeated for each disk connected. At this point, there may be a number of drivers in memory, but there are no volumes, since none have been mounted yet. Generally there is one driver per disk, but some drivers can handle more than one disk at a time.

2) Decide which disk is to be the startup disk:

a) The next stage is to determine which volume will become the startup disk. If there is a floppy available it will always be the startup disk. During this process the disk chosen as the startup disk is not known to be valid. The System file and boot blocks are checked later.
b) The standard HD-20 is connected to the system in a fashion that is very similar to a floppy, so if a bootable HD-20 is connected it will be the startup disk.
c) There is no search for floppy devices like there is for SCSI disks since the driver for the floppies will post a Disk Inserted event when it detects a floppy in the drive. The first floppy device that is found will be used as the startup disk. If there are multiple floppy devices, the others will be mounted by the Finder, not at boot time. The SCSI devices that are online are not mounted at this time, either. There is a pending Disk Inserted event for each disk that will be handled by the Finder.
d) At boot time, there is only one volume that is mounted (during execution of the Macintosh boot blocks). The others will be mounted when their Disk Inserted event is processed at a `GetNextEvent` call.
e) On the new Control Panel there is a Control Device (cdev) called the Startup Device. This Startup Device cdev allows the user to choose which device the system should try to boot from first. This can only be used on the Macintosh II and SE. The drive number, driver reference number, and driver OS type are stored in parameter RAM to allow a chosen device to be the boot disk. The floppy drives will still have precedence over the SCSI devices. The standard HD-20 can be chosen as the Startup Device as well, since it uses a different driver reference number. If the drive number that is stored as the Startup Device is invalid, or had a read/write error, then another disk in the chain will be chosen as the next bootable candidate. Remember that there is only one boot/startup/system disk, and it is the only one that is explicitly mounted at boot time. All other devices in the system will be handled once the system is booted.

3) Execute the Macintosh boot blocks:

a) Once the Startup Disk has been chosen (whether floppy, SCSI or other disk) then it is time to read the Macintosh boot blocks off of blocks 0 and 1 of the volume. Those boot blocks determine various parameters in the system, such as whether a Macsbug-like debugger will be loaded, the name of the startup program (not always the Finder), how big to make the event queue, how big to make the system heap, and so on. They also contain a signature identifying them as Macintosh boot blocks, and a version number to differentiate between different boot blocks.

b) After the boot blocks are read and the signature verified, the smiling Macintosh is displayed on the screen. The smiling Macintosh basically means that valid Macintosh boot blocks were found.

c) On 64K ROMs the boot blocks are executed by jumping to the code that follows the header information in boot block 0. On the newer Macintoshes the boot block version number is checked, and if it is 'old' the boot blocks will be skipped. The same code that would have been found in the boot blocks is found in the ROM itself. Regardless of which kind of Macintosh it is, the following steps apply. For the newer Macintoshes the boot blocks are usually used only for the parameters stored in the header.

d) Do the `PBMountVol` on the chosen startup volume. If `PBMountVol` fails, the process starts over at the point where a startup disk is being chosen (step 2 above). The failing volume is marked out of the list of candidates so that it won't be used again.

e) Find the System file and create a Working Directory, if needed, for the System folder. This is only done for HFS volumes of course, and the directory ID is set to the blessed folder. The blessed folder is saved in the volume header as part of the `FinderInfo` field. See Technical Note #67 for more information on the blessed folder. If the directory ID is wrong, the System file won't be found, causing it to start over again (at step 2 above). If the Working Directory was created successfully, that `WDRefNum` is set as the default volume with `SetVol`.

f) The System file is opened with `OpenResFile`. If the file could not be opened, the process starts over again at the point where a suitable boot device is being chosen (step 2 again).

g) The Startup Screen is loaded and displayed. If there was no Startup Screen, the normal "Welcome to Macintosh" message will be displayed. The Startup Screen or "Welcome..." means that the System file was found and opened successfully. On the Macintosh Plus and 64K ROM machines, the Startup Screen is displayed before the System file is opened. (reverse steps f and g)

h) The debugger and disassembler are installed if found. The names of the debugger and disassembler are found in the header of the boot blocks and are usually Macsbug and Disassembler respectively.

i) The **data** fork of the System file is opened and executed. The data fork contains code to read in the PTCH resources which patch the ROM.

j) The INITs that are in the System file are executed. The last INIT is INIT 31 which then looks in the System Folder for other INITs to be executed.

k) The file specified by the boot blocks as the startup application (Set Startup at the Finder) is found on the volume, using another field in the `FinderInfo` field of the volume header in order to get the Directory ID. If the file exists, it is launched. If not, the Finder is launched. If the Finder is not found, `SysError` is called with error code of 41 which is the "Can't launch Finder" alert.

4) The Finder uses the Desktop file on the volume to draw the desktop.

If the startup application was the Finder, it opens the Desktop file on the startup volume in order to draw the desktop. When it finishes with the startup volume, it calls `GetNextEvent`. If there are any pending Disk Inserted events, the volume specified is mounted (by the ROM) and the result passed to the Finder. If `PBMountVol` failed for any reason, the bad result will be passed to the Finder. At that point the Finder would put up the "This disk is damaged" alert and ask if the volume should be initialized or ejected. If ejected, the driver for that volume still exists, but the volume is unmounted. For each volume that the Finder sees, it opens the Desktop file on the volume to get the information that it needs to build the desktop. If the Desktop file was not found on a volume, it is created. If there are any errors while creating or using the Desktop file, the Finder will display the "This disk needs minor repairs" message. If the OK button is clicked, the Finder will delete the old file and create a new one. If that fails, the volume is unmounted and deemed unusable by the Finder. This happens if the disk is locked, or too full to add a Desktop file. If that was the startup volume, the computer is rebooted since it was forced to unmount the startup volume, and cannot run if there is no startup volume.

If you follow the previous sequence closely, you can predict what errors are causing a given end result. For example, if you have the effect where the smiley Macintosh appears, but immediately goes away and the disk does not boot, you can look through the sequence to see what might be going wrong. In this case, we know that the boot blocks were found on our startup volume, since the smiley Macintosh was displayed. We know that the System file was not found, or failed to open, since we never got the Welcome message. This usually calls for throwing away all of the System Folders on the volume, and starting again with a new System Folder to fix the problem. If there is more than one System Folder on a volume it is possible to confuse the system.

Other tidbits of information that may be useful (in no particular order) some which will be mentioned in the step-by-step operation below:

1) The SCSI cables have a lot of wires in them, and are rather bulky because of it. It is best to avoid bending the cables too much or too often, since the wires inside will break if overstressed. Don't put wild kinks in the cable in order to make it fit behind the Macintosh.
2) If there is no default volume stored in the parameter RAM with the Startup Device cdev, then the first drive that is in the drive queue will be the Startup Device. Since SCSI drives are added in highest ID order, that means the larger SCSI IDs will have a higher 'priority'. Macintosh IIs will default to the internal hard disk.
3) If the parameter RAM is trashed for some reason, the boot process can fail since a driver OS type is stored as well. If the OS type is wrong, the ROM will skip that driver, making the disk unbootable. On the Macintosh II/SE, the battery is no longer removable to fix parameter RAM problems. To correct this problem the Control Panel now has a feature that will allow you to clear parameter RAM. Holding down the Option-Command-Shift keys while opening the Control Panel will reset parameter RAM, forcing it to be rebuilt and therefore losing all of your settings, but possibly fixing some booting problems.

4) The Macintosh II and SE both have a new feature that will allow you to skip having the any hard disk mounted. Holding down the Option-Command-Shift-Delete combination will have the startup code skip the SCSI hard disks on the system. This can be useful if you are booting an old System file that does not understand HFS disks (like System 2.0/Finder 4.1), and want to avoid having your hard disks on line while you do something shaky. With external hard disks it is easier to just turn them off, but with internal disks it is not so easy.
5) Since the parameter RAM can be trashed in a manner that makes it impossible to boot a volume (looking for the wrong OS type), a new feature was added to the HD SC Setup program to have it fix this problem as well. If you have version 1.3 or greater, the parameter RAM bytes that determine booting will be reset to fix some boot problems that occur. The parameter RAM is fixed when the Update button is clicked. This does not invalidate the rest of parameter RAM, it merely fixes the bytes used for the Startup Device.
6) When the Finder copies a new System Folder onto a disk that does not already have a System Folder, that new folder will become the blessed folder. Its Directory ID will be saved in the volume header. In addition, the Macintosh boot blocks will be copied from the current startup device to the destination device. This is the best way to fix System Folder or Macintosh boot block problems. In order for the blessed folder to be set correctly, all System Folders on the volume should be deleted before copying the new folder there.
7) If the Desktop file is damaged for whatever reason, it can be deleted with a number of programs. This will force the Finder to rebuild it from scratch. You can also have the Finder rebuild the Desktop file by holding down the Option-Command keys when the Finder is launched. When the Desktop file is rebuilt you lose the Finder Comments in the Get Info boxes.
8) On the 64K ROMs, whenever something goes wrong during booting (like System file not found, bad boot blocks, and so on) the Sad Mac Icon is displayed. Starting with the 128K ROMs, whenever something goes wrong the ROM jumps back to the start to try to find another disk to use.

Bo3b's Boot Repair

This section will detail step-by-step processes that can be used to fix some common booting and volume problems. It is not intended to cover every possible case. The purpose of the preceding sections was to give you the information that will allow you to figure out what might be going wrong.

For most hard disk users, it is not sufficient to merely have the device running. It is generally a good idea to make the system as robust as possible in order to avoid some of the problems that might cause a volume to become wholly unreadable. The ultimate fix is to reinitialize the volume from scratch and rebuild the volume with the Finder or a restore operation that uses the File Manager. This is guaranteed to fix anything except hardware problems, and will give you the most solid system. If your system is acting funny, you can try the following sequence that is the next best thing to initializing the disk. This sequence will not make you rebuild the disk, but can be fooled by some disk problems. If everything passes, then the disk is in good shape; maybe not perfect, but good.

1) Power down the entire system, including the hard disk that is suspect.
2) Run the HD SC Setup program (or equivalent) and Update the drivers on the disk. For HD SC, this also fixes the parameter RAM. For non-Apple drives, the parameter RAM can be reset with the Control Panel.
3) Run the Test Disk option in HD SC Setup (or equivalent). If the test fails, reinitialize the volume, since it is not worth risking future problems.
4) Run the Disk First Aid utility. This utility will work on all HFS volumes. Have it check the volume for consistency. If it reports any errors, you can have it fix the problem, but the safest tack is to reinitialize. There are some problems that Disk First Aid won't catch. If Disk First Aid says the volume cannot be verified, it is time to reinitialize.
5) Rebuild the Desktop file by holding down Option-Command when returning to the Finder.

If you can successfully perform all of these steps, the volume will be as solid as it can get without reinitializing the disk. If things are still funny, it is time to take the last recourse, reinitialize.

Based on the previous sections, it is now time to go through all of the Nasties to give a step-by-step sequence for fixing these problems.

1) The data is intact, but the hard disk won't boot.

This is for the case where the volume won't boot, but if the computer is booted with a floppy disk the volume shows up at the desktop and can run normally. For this case, we know that the driver is being loaded and working, since the volume shows up at the desktop. The volume is also mountable, since it shows up with no problem. This implies that the Macintosh boot blocks are wrong, or the blessed folder is wrong. Clues such as the smiling Macintosh can tell you how far the process got before it failed. For example, if the smiling Macintosh never appeared, we know that Macintosh boot blocks were not read successfully. When the volume is fixed and bootable, it would be a good idea to go through the steps above to make the volume as solid as possible.

The sequence to follow:

a) Power down the entire computer, including the hard disk. Try to boot again. If it works, you are done.
b) Use the Control Panel's Startup Device to set the hard disk as the Startup Device. This will also reset some of the bytes in parameter RAM. Try rebooting to see if it has fixed the problem.
c) Run HD SC Setup (or equivalent) and perform the Update Drivers procedure. In the HD SC Setup case this will also rewrite the parameter RAM. If you are not using HD SC Setup, blast the parameter RAM with the Control Panel. Try rebooting.
d) Delete **all** System Folders from the hard disk. Using Find File or something similar, be sure that there are no stray copies of the System or Finder buried in some long lost folder. Copy a new System Folder to the volume, using the Finder. This process will fix bad boot blocks, as well as a bad blessed folder. Try rebooting.
e) If it still won't boot, there is something very strange happening. Whenever things get too weird it is usually time to start over: reinitialize.

2) The disk won't boot and only shows up after running HD SC Setup.

The disk does not even show up at the Finder when the system is booted with a floppy. After running the HD SC Setup (or equivalent) the volume will appear on the desktop and be usable. The HD SC Setup and most similar utilities will do an explicit `PBMountVol` of the volume in order to make the volume usable. Since the volume does not show up at the Finder at first, this implies that the driver itself is not getting loaded or is working improperly, since there was no Disk Inserted Event for the Finder to use.

The sequence:

a) Power down completely, including the hard disk.
b) Run HD SC Setup (or equivalent) and Update the Drivers. For non-Apple drives, update the drivers on the volume (this rewrites the SCSI partition information as well) using the utility that came with the disk. Reset the parameter RAM using the Control Panel.
c) If it still cannot be booted or does not show up at the Finder after booting with a floppy, the volume is too weird and should be reinitialized.

3) The disk will boot but hangs part way through the boot process.

This is when you can see the volume is being accessed by the run light (LED) on the front panel, and the booting seems to work but never makes it to the Finder. This implies that all is well until the System tries to actually launch the Finder or Startup Application. It could also be that the System file is causing something to hang.

The sequence:

a) Power down completely.
b) Boot with a floppy so that the floppy is the startup disk and the volume in question can be seen at the Finder.
c) Delete all System Folders on the hard disk. Put a new System Folder on the disk. This will presumably fix a corrupted System file.
d) If still funky, show the disk who's boss.

4) There are data errors while the disk is running.

This case usually evidences itself by messages at the Finder when trying to copy files. Messages like "The file ^0 could not be read and was skipped" usually mean that the drive is passing back I/O errors. This usually means that there is a hardware failure, but it can occasionally be caused by bad sectors on the disk itself. If the sectors are actually bad, it is generally necessary to reinitialize the volume.

The sequence:

a) Power down completely. Reboot and see if the same file gives the same error.
b) Run the HD SC Setup (or utility that came with your drive) and perform the Test operation. This will fail if there are bad blocks on the device. If there are bad blocks, it is necessary to reinitialize the volume.
c) Check the SCSI terminators to be sure they are plugged in correctly. There can be no more than two terminators on the bus. If you have more than one SCSI drive you must have two terminators. If you only have one drive, use a single terminator. If you have more than one drive, the two terminators should be on opposite ends of the chain. The idea is to terminate both ends of this wire that goes through all of the devices. If you have a Macintosh II or SE with an internal drive, that drive will already have a terminator inside the Macintosh at the front of the cable.

d) Make sure the SCSI cables you are using are OK, by swapping them with known good ones. If the problem disappears, the cable is suspect.
e) Swap the terminators in use with known good ones to be sure they are OK.
f) Try the drive and cable on a different Macintosh to be sure the Macintosh is OK.

5) The disk is very slow returning to the Finder.

If the computer has gotten slower with age, it is probably due to a problem with the Desktop file. If a volume has been used for a long time, the Desktop file can grow to be very large (Hundreds of K). Reading and using a file that big can slow down the Finder when it is drawing the desktop. If you have a large number of files in the root directory, this will also slow the computer down. A large number (500-1000) of files in a given folder can cause performance problems as well. If a volume has been used for a long time, it can also have become fragmented.

The sequence:

a) Rebuild the Desktop file and see if it gets faster.
b) Look for large numbers of files in a given directory and break them up into other folders if needed.
c) Run Disk First Aid to be sure the volume is not damaged.
d) Reinitialize the volume and restore the data using File Manager calls to fix a fragmentation problem. Using the Finder, or a backup program that reads and writes files is a way to use only File Manager calls. You cannot fix a fragmentation problem by doing an image backup and restore.

6) The computer crashes or hangs when returning to the Finder.

This can happen if the Desktop file becomes corrupted. There are occasions when this can happen if the HFS structures on the volume are damaged.

The sequence:

a) Rebuild the Desktop file.
b) Run Disk First Aid to be sure the volume is not damaged; a boot floppy with the Set Startup set to Disk First Aid can allow you to test a volume that cannot be displayed at the Finder.
c) The path of ultimate recourse if nothing else seems wrong with the volume.

7) The disk appears in a "This disk is bad" dialog.

This is the worst of the possible errors that generally happen to hard disks. If the message is "This disk is bad" or "This is not a Macintosh disk", the HFS structures on the volume have been damaged. In particular, the Master Directory block on the volume has been damaged. The driver and SCSI partition information are probably OK, since this dialog shows up when the Finder tries to mount a damaged volume. This means that the `PBMountVol` call failed. Don't click the Initialize button unless you are sure you want the volume to be erased. In these cases, it is nearly always better to just reinitialize the volume after you have saved whatever information you can.

The sequence:

a) Power down completely. Occasionally the controller in the hard disk itself can crash.
b) Run Disk First Aid. For these cases, it is usually necessary to create a boot floppy with Set Startup set to Disk First Aid. When the floppy is booted, Disk First Aid will be run before the Disk Inserted events are processed. When Disk First Aid sees the Disk Inserted event it will check the result from the `PBMountVol` and still allow you to test the volume, even if it can't be mounted.
c) If Disk First Aid cannot repair the disk, it might be worth writing a simple program to call the driver to read and write blocks. There is a copy of the Master Directory Block on the end of the volume, and the volume can sometimes be fixed by copying that block over a damaged block in sector 2. You can write a program that will find out how big the volume is by looking in the Drive Queue Element for the volume, reading the block that is one sector from the end (N-1), and writing that copy over sector 2. At this point, the volume is probably inconsistent, but it may allow you to use it long enough to get information off of it. It is sometimes possible to have Disk First Aid repair the volume at this point as well. Copying the sectors can also be done with sector edit utilities, if you can get them to recognize the volume at all.
d) If making a new copy of sector 2 does not work, but the driver is still being loaded at boot time, it is possible to write a program that will read sectors from the disk looking for information that you might need. You can have a reader program go through blocks looking for a specific pattern, like a known file name. This is usually done in desperation, but sometimes there is no other choice. If the data desired can be found in some form, it can sometimes be massaged back to a useful form much easier than recreating it.
e) Sometimes the volume will be so badly damaged that the SCSI partition information is also damaged and cannot be fixed with the Update in the hard disk utility. In this case, it is usually still possible to perform direct SCSI reads, without going through the driver. Using the driver is preferable, since it knows how to talk to the drive better than you would, but sometimes the driver is not available. Using direct SCSI reads should be a last ditch effort since the SCSI Manager can be very challenging to use. This should only be used if there is irreplaceable data on the volume that cannot be read by any other means.
f) Even if the volume is recovered, it still should be reinitialized (after the data is recovered) to be sure that any hidden damage is repaired.

8) The disk never shows up at all.

The disk appears to be missing. The volume does not show up at the Finder, and does not show up in HD SC Setup. At boot time the access light (LED) does not flash. This is usually a hardware problem as well. The drive is not responding to SCSI requests at all, so the system cannot tell a drive is attached.

The sequence:

a) Power down the system, including the hard disk.
b) Make sure that the SCSI ID on the drive does not conflict with any other in the system, including the Macintosh, which is ID 7. (If you have an internal hard drive, it should be ID 0.)

c) Check the SCSI terminators to be sure they are plugged in correctly. There can be no more than two terminators on the bus. If you have more than one SCSI drive you must have two terminators. If you only have one drive, you should use a single terminator. If you have more than one drive, the two terminators should be on opposite ends of the chain. The idea is to terminate both ends of this wire that goes through all of the devices. If you have a Macintosh II or SE with an internal drive, that drive will already have one terminator inside the Macintosh at the front of the cable.
d) Make sure the SCSI cables you are using are OK, by swapping them with known good ones.
e) Swap the terminators in use with known good ones to be sure they are OK.
f) Try the drive and cable on a different Macintosh to be sure the Macintosh is OK.

These boots are made for wokking

Remember, the goal here is to make the system be as stable as possible. If things are acting strange, it doesn't hurt to go through the entire process of testing the drive. The test procedure takes a little time but is non-destructive for the data that is there. If something catastrophic has happened to the disk, it is better to spend some time backing up the data, initializing the volume, and restoring the data than it is to lose some work later on due to some other permutation of the same problem. Unless you are sure that the volume is in an undamaged state, you are better off using a file-by-file backup operation than an image backup, since the image backup will copy any damage as well as the data.

If there are situations that you run into that are not covered by this technical note, please let us know so that they can added.

If this technical note helps even one person save some data that would otherwise be lost, it will have been worthwhile. Hope it helps.



#135: Getting through CUSToms

See also: Technical Note #88—Signals
Technical Note #110—MPW: Writing Standalone Code

Written by: Rick Blair
Updated:

July 1, 1987
March 1, 1988

This technical note provides a way for developers to allow sophisticated users to add code to an off-the-shelf application. Using this scheme, the user can easily install the code module; the application has to know how to call it and, optionally, be able to respond to a set of predefined calls from the custom package.

Note

The following code makes heavy use of features of the Macintosh Programmer's Workshop. It also assumes a basic familiarity with the standard Sample program included with MPW. The Pascal code (which is here only as an example implementation of the mechanism) is presented as only those sections which *differ* from Sample.p. The assembly language code also includes MPW-only features, such as record templates. Some of these are explained in Technical Note #88, "Signals."

In addition, since the order in which parameters to various routines are passed is critical, special care will have to be taken in writing interfaces for use with C. It is probably best to declare them as Pascal in the C source.

Concepts

Basically, we create a code resource of type CUST with an entry point at the beginning which takes several parameters on the stack; this code is reached via a dispatching routine which is written in assembly language.

The data passed on the stack to this dispatcher includes:

- a selector (to specify the operation desired)
- the address of a section of application globals (for communication back and forth between the application and the module when the stack parameters are insufficient)
- a handle which references the custom code resource on the stack.

Other parameters may be added (as long as they are pushed on the stack before the required ones) if desired. Since these extra parameters would **always** have to be included in any calls to a given package, it might be more convenient to use the application global space area which is accessed through the `appaddr` parameter.

Template

Your application must contain the following global data and procedure declarations to support this model:

```
VAR
      custhandle: Handle;

      {the following globals constitute the data known to the custom code}
      appdispatch: ProcPtr; {address of dispatch routine custom code can call}
      {examples of further application globals for the custom package:}
      (*
      paramptr: Ptr; {general pointer used as param. to appdispatch code}
      paramword1: INTEGER;
      paramword2: INTEGER;
      CUSTerr: INTEGER;
      *)
      {any other globals the module should get at}

     {the two assembly language glue routines which are linked into the
      application}
     PROCEDURE CustomInit(resID: INTEGER; VAR custhandle: Handle);
      EXTERNAL; {the routine used to set up the custhandle resource handle}

     PROCEDURE CustomCall({application & package-specific paramters}
                 selector: INTEGER; appaddr: UNIV Ptr; ourhandle: Handle);
      EXTERNAL; {this is the code dispatcher}

    {this is called by the custom package to perform a service which is more
     easily provided by the application; since we pass a pointer to it to the
     package, CustDispatch must be at the outermost nesting level in the main
     segment}
     PROCEDURE CustDispatch(selector: INTEGER);

      BEGIN
         CASE selector OF
          {.
           .
           .}
         END; {CASE}
      END; {CustDispatch}

    {your initialization code should contain the following:}

      {Custom package initialization stuff}
      appdispatch := @CustDispatch; {put pointer where the package can see it}
      CustomInit(69,custhandle);    {our CUST resource has ID = 69}

    {then whenever you want to invoke the package you use CustomCall}
```

You must also assemble CustomInit and CustomCall and link them with into your application. The custom package itself can be written in any language which can produce stand-alone code. See Technical Note #110 for how to write stand-alone code in MPW Pascal.

The example

`CustomCall` is only referenced once in this example. When a variety of unrelated functions are provided, however, it is more convenient to provide a separate interfacing procedure to invoke each one and have them make their own `CustomCall` calls.

Note that this example is somewhat contrived; you probably wouldn't "externalize" the code for finding a word or sequence of characters like this. This is an idealized situation. More realistic uses would be: to add-on special routines to a database to perform custom calculations or the like; allow for localization when code is required (and hooks aren't already provided); let documents carry around code which may vary among software versions, etc. so that older documents would be able to work alongside the new ones, etc.

What it does

We simply add a new menu to the sample program which allows **Find** by characters or word. We just pass the menu item to the package and let it do the finding; it then calls back to the application dispatch routine to highlight text or display the "not found" message.

The Pascal source for the example application appears first:

```
{$R-}
{$D+}
PROGRAM P;

USES
 {$LOAD ::PInterfaces:most.dump}
 Memtypes,Quickdraw,OSIntf,ToolIntf,PackIntf {,MacPrint}
 {$LOAD}
 , {$U ErrSignal.p} ErrSignal;

CONST
 appleID = 128; {resource IDs/menu IDs for Apple, File and Edit menus}
 fileID = 129;
 editID = 130;
 findID = 131;

 appleM = 1; {index for each menu in myMenus (array of menu handles)}
 fileM = 2;
 editM = 3;
 findM = 4;

 menuCount = 4; {total number of menus}
```

```
windowID = 128; {resource ID for application's window}

undoCommand = 1; {menu item numbers identifying commands in Edit menu}
cutCommand = 3;
copyCommand = 4;
pasteCommand = 5;
clearCommand = 6;

findcharsCommand = 1; {menu items for Custom menu}
findwordCommand = 2;

aboutMeCommand = 1; {menu item in apple menu for About sample item}

aboutMeDLOG = 128;
findDLOG = 129;
infoDLOG = 130;

{application dispatching code selectors}
hilightSel = 0;
notifySel = 1;

VAR
•
•
•
errCode: INTEGER;
dlogString: Str255;
custhandle: Handle;

{here is the area known to the custom code}
appdispatch: ProcPtr; {address of dispatch routine custom code can call}
{examples of further application globals for the custom package}
paramptr: Ptr; {general pointer used as param. to appdispatch code}
paramword1: INTEGER;
paramword2: INTEGER;
{any other globals the module should get at}


PROCEDURE CustomInit(resID: INTEGER; VAR custhandle: Handle);
EXTERNAL; {the routine used to set up the custhandle resource handle}

PROCEDURE CustomCall(text: Ptr; count: INTEGER; findstr: StringPtr;
          selector: INTEGER; appaddr: UNIV Ptr; ourhandle: Handle);
EXTERNAL; {this is the code dispatcher}


{this will do the "about" dialog and the info dialog requested by the
custom pack.}

PROCEDURE ShowADialog(meDlog: INTEGER);

CONST
   okButton = 1;
   authorItem = 2;
   languageItem = 3;
   infoItem = 2;
```

```
VAR
   itemHit,itemType: INTEGER;
   itemHdl: Handle;
   itemRect: Rect;
   theDialog: DialogPtr;

BEGIN
   theDialog := GetNewDialog(meDlog,NIL,WindowPtr( - 1));

   CASE meDlog OF
      aboutMeDLOG: BEGIN
      GetDitem(theDialog,authorItem,itemType,itemHdl,itemRect);
         SetIText(itemHdl,'Ming The Vaseless');
         GetDitem(theDialog,languageItem,itemType,itemHdl,itemRect);
         SetIText(itemHdl,'Pascal et al');
      END;

          infoDLOG: BEGIN {display the message requested by the custom
                           package}
         GetDitem(theDialog,infoItem,itemType,itemHdl,itemRect);
         SetIText(itemHdl,StringPtr(paramptr)^);
      END;
   END; {CASE}

   REPEAT
      ModalDialog(NIL,itemHit)
   UNTIL (itemHit = okButton);

   CloseDialog(theDialog);
END; {of ShowADialog}


{this will put up the Find dialog to allow the user to type in the
characters to search for}
FUNCTION DoCustomDialog: BOOLEAN;

CONST
   okButton = 1;
   cancelButton = 2;
   fixedItem = 3;
   editItem = 4;

VAR
   itemHit,itemType: INTEGER;
   itemHdl: Handle;
   itemRect: Rect;
   theDialog: DialogPtr;

BEGIN
   theDialog := GetNewDialog(findDLOG,NIL,WindowPtr( - 1));
   GetDitem(theDialog,editItem,itemType,itemHdl,itemRect);
   SetIText(itemHdl,dlogString);
   TESetSelect(0,MAXINT,DialogPeek(theDialog)^.textH);
```

```
    REPEAT
       ModalDialog(NIL,itemHit)
    UNTIL (itemHit IN [okButton,cancelButton]);
    GetIText(itemHdl,dlogString);
    DoCustomDialog := itemHit = okButton;

    CloseDialog(theDialog);
 END; {of DoCustomDialog}


PROCEDURE DoCommand(mResult: LONGINT);
•
•
•
(* partial procedure fragment *)

{here is one of the case sections for the DoCommand procedure}

       findID:
          IF DoCustomDialog THEN
             BEGIN
             MoveHHi(Handle(textH)); {stop it from fragmenting the heap}
             WITH textH^^ DO BEGIN
              HLock(hText); {since we don't know what the package might
                                          be up to}
              {now call the package to find characters or words}
               CustomCall(POINTER(ORD(hText^) + selEnd),
                teLength - selEnd, @dlogString, theItem, @appdispatch,
                custhandle);
               HUnLock(textH^^.hText);
             END; {WITH}
             END;

    END; {OF menu CASE} {to indicate completion of command,}
    HiliteMenu(0); {call Menu Manager to unhighlight }
    {menu title (highlighted by     }
    {MenuSelect)          }
 END; {OF DoCommand}

{this is called by the custom package to set the new selection or display a
  message; it must be in CODE 1 at the outermost lexical level}
 PROCEDURE CustDispatch(selector: INTEGER);

 BEGIN
    CASE selector OF
       hilightSel: {hilight the characters selected by the custom pack.}
      {paramptr=pointer to text to select, paramword1&paramword2=start,end
             chars}
          WITH textH^^ DO
           {we'll subtract the start of text from paramptr to get the base
              offset…}
                TESetSelect(ORD(paramptr) - StripAddress (ORD(hText^)) +
                    paramword1, ORD(paramptr) - StripAddress (ORD(hText^))
                     + paramword2,textH);
```

```
        notifySel: {put up message per request from custom pack.}
        {paramptr points to string to display}
           ShowADialog(infoDLOG);

    END; {CASE}
 END; {CustDispatch}

BEGIN {main program}
 { Initialization }
 InitGraf(@thePort); {initialize QuickDraw}
 InitFonts; {initialize Font Manager}
 FlushEvents(everyEvent - diskMask,0); {call OS Event Mgr to discard
                                        non-disk-inserted events}
 InitWindows; {initialize Window Manager}
 InitMenus; {initialize Menu Manager}
 TEInit; {initialize TextEdit}
 InitDialogs(NIL); {initialize Dialog Manager}
 InitCursor; {call QuickDraw to make cursor (pointer) an arrow}

 InitSignals;
 errCode := CatchSignal;
 IF errCode <> 0 THEN BEGIN
    Debugger;
    Exit(P);
 END;

 SetUpMenus; {set up menus and menu bar}
 UnLoadSeg(@SetUpMenus); {remove the once-only code}

 {Custom package initialization stuff}
 appdispatch := @CustDispatch;
CustomInit(69,custhandle); {should test custhandle for NIL and alert
                            the user}
 dlogString := '';
...
{etc. with the rest of initialization and the main event loop}
END.

; now for the assembly language code
; first, the dispatching and initializing code that must be linked into
; the application
; CustomCalling
; Custom packages initializing and dispatching
;
;   Rick Blair      May, 1987
;
;           PRINT   OFF
;           INCLUDE 'Traps.a'
;           INCLUDE 'ToolEqu.a'
;           INCLUDE 'QuickEqu.a'
;           INCLUDE 'SysEqu.a'
;           PRINT   ON

            LOAD    'most.dmp'  ; from a dump of the files above

appdata     EQU                 12
```

```
;Initialize a custom module
; Pascal call format:
;  CustomInit(resID:INTEGER;VAR custhandle:Handle);
;
; This will load the CUST module with the given resource ID, install a
; handle to it in custhandle, and set the module's appdata pointer to
; point to the address appaddr.
;
resID       EQU                 8
custhandle  EQU                 4

CustomInit  PROC    EXPORT
            SUBQ.L  #4,A7       ;make room for handle from GetResource
            MOVE.L  #'CUST',-(A7)
            MOVE.W  resID+8(A7),-(A7);resource ID
            _GetResource
            MOVE.L  (A7)+,A0
            MOVE.L  custhandle(A7),A1
            MOVE.L  A0,(A1)     ;store handle in app's custhandle global
;(return with nil handle if GR failed)
            MOVE.L  (A7),A0     ;get return address
            ADD.L   #10,A7      ;strip everything
            JMP     (A0)        ;adieu


;Call a custom module
;Pascal format:
; CustomCall( {parameters as desired} selector: INTEGER; appaddr: Ptr;
;              module: Handle);
;
;This will call the code whose handle is passed on the stack. If the
;application was written in assembly language you would just
;dereference the handle and call it directly (you wouldn't need this at
;          all).
;
CustomCall  PROC    EXPORT
            IMPORT  Signal
            MOVE.L  4(A7),A0    ;get handle
            MOVE.L  (A0),D0
            BNE.S   @0          ;if hasna' been purged, ga' ahead
            MOVE.L  A0,-(A7)    ;push handle
            _LoadResource
            MOVE.W  ResErr,-(A7)
            JSR     Signal      ;Signal is a NOP if a zero is passed to it
            MOVE.L  4(A7),A0    ;handle again
; we don't lock the handle here (we can't save it so we can unlock it
;  later), so it's up to the package to lock/unlock itself
@0          MOVE.L  (A0),A0     ;dereference
            JMP     (A0)        ;call CUST code

            END
```

```
; here is the module for the custom package itself

; CustomPack
; Example custom code package
;
;   Rick Blair      May, 1987
;
; This demonstrates the recommend structure of a code module which a
; sophisticated user could add to an existing application which supported
; this mechanism. Aside from allowing for multiple routines within the
; module (via a selector), provision is made for calling a routine
; dispatcher within the application itself.

;Finding text
;We support a call to find a string anywhere within a block of text
; (selector=0), and one to find the string only as a separate "word"
; with spaces around it (selector=1).
;PROCEDURE CustomCall(text:Ptr; count:INTEGER; findstr:^STRING;
;                     selector:INTEGER; appaddr: UNIV Ptr; ourhandle:Handle);
;Rather than return a result indicating whether they succeeded or not,
;these routines take whatever action is appropriate (the application
;may not even know what these routines actually do).
;Once a call succeeds or fails, it then takes action by making a call to
;one of the services provided by the application. In this case the two
;functions provided are just what we need; the ability to select text and
;the ability to put up a message saying "Text not found".

            STRING  ASIS

;           PRINT   OFF
;           INCLUDE 'Traps.a'
;           INCLUDE 'ToolEqu.a'
;           INCLUDE 'QuickEqu.a'
;           INCLUDE 'SysEqu.a'
;           PRINT   ON

            LOAD    'most.dmp'  ; from a dump of the files above

CustPack    PROC    EXPORT

            BRA.S   Entry       ;skip header

            DC.W    0           ;flags
            DC.B    'CUST'      ;custom add-on code module
            DC.W    69          ;resource ID (picked by Mr. Peabody &
                                ; Sherman)
            DC.W    $10         ;version 1.0


StackFrame  RECORD  {A6Link},DECR
paramsize   EQU     *-8
;       call-specific parameters… (optional)
text        DS.L    1           ;pointer to text block
count       DS.W    1           ;word count of characters in text
findstr     DS.L    1           ;pointer to p-string to find
;       selector(word, optional - you might only have 1 call)
selector    DS.W    1
```

```
fcharsCmd   EQU     1           ; selector for "find characters"
fwordCmd    EQU     2           ; selector for "find word"
;       pointer to app. globals(long)
appaddr     DS.L    1
;       handle to this resource(long)
ourhandle   DS.L    1
;       TOS:return address (long)
return      DS.L    1
;the stack link is built off the origin of the saved old A6 on the stack
A6Link      DS.L    1
LocalSize   EQU     *
            ENDR

;offsets into our application globals area
AppGlobals  RECORD  {appdispatch},DECR
appdispatchDS.L     1
paramptr    DS.L    1
paramword1  DS.W    1
paramword2  DS.W    1
;CUSTerr    DS.W    1           ;if we had possible errors
            ENDR

Entry
            WITH    StackFrame,AppGlobals
            LINK    A6,#LocalSize
;           MOVEM.L …           ;we'd save any non-trashable regs here
;first lock us down…
            MOVE.L  ourhandle(A6),A0
            _HLock

            MOVE.W  selector(A6),D0
            CMP.W   #fcharsCmd,D0
            BEQ.S   charfind    ;go find characters
            CMP.W   #fwordCmd,D0
            BEQ.S   wordfind    ;go find a word
;well, M. App didn't call us with a selector we know, so…

;unlock ourselves, clean up, return
; (if we wanted to return an error code we could stuff it into the app.
;  global area)
duhn        MOVE.L  ourhandle(A6),A0
            _HUnLock
;           MOVEM.L …           ;restore any registers here
            UNLK    A6
            MOVE.L  (A7)+,A0    ;return address
            ADD.L   #paramsize,A7;strip parameters
            JMP     (A0)

;selector codes for calls to application
hilight     EQU     0           ;highlight characters, please
notify      EQU     1           ;beep a little

;find the string "findstr" anywhere in the block "text"
charfind
            JSR     findchars   ;see if findstr is anywhere in text
            BEQ.S   nofind      ;if not then skip
            JSR     calcsels    ;compute selstart and selend
didfind     MOVE.L  appaddr(A6),A0 ;get pointer to appl. globals area
```

```
            MOVE.L  text(A6),paramptr(A0) ;setup text pointer and…
            MOVE.W  D0,paramword1(A0) ;start character position,
            MOVE.W  D1,paramword2(A0) ;end character position
            MOVE.W  #hilight,-(A7) ;pass proper selector
goapp       MOVE.L  appdispatch(A0),A0 ;get dispatch address
            JSR     (A0)        ;call the application to select the range
            BRA.S   duhn        ;return to application (dejà vu)

nofind      MOVE.L  appaddr(A6),A0  ;get pointer to appl. globals area
            LEA     oopstring,A1    ;get pointer to "Not found" message
            MOVE.L  A1,paramptr(A0) ;put string pointer in "paramptr"
            MOVE.W  #notify,-(A7) ;tell app. to display message
            BRA.S   goapp

;figure selstart and selend
calcsels    NEG.W   D0          ;negate # characters unskipped in text
            SUBQ.W  #1,D0       ;include 1st character
            ADD.W   count(A6),D0;compute 1st character position for select
            MOVE.L  findstr(A6),A1
            MOVE.B  (A1),D1     ;get length of string
            EXT.W   D1
            ADD.W   D0,D1       ;compute last char. pos. for select
            RTS

;find the characters, but only if surrounded by space (including end or
; beg.)
;we could extend the test to check for other delimiters (";",".",etc.)
wordfind
            JSR                 findchars
wloop       BEQ.S   nofind
            MOVE.W  D0,D2       ;save count of text remaining
            JSR     calcsels    ;figure start and end offsets
            MOVE.L  text(A6),A1 ;point to text
            TST.W   D0          ;start=beginning of text?
            BEQ.S   @0          ;yep, so it passes
            CMP.B   #' ',-1(A1,D0) ;preceded by a space?
            BNE.S   @1          ;nope, keep looking
@0          CMP.W   count(A6),D1 ;D1=length of text?
            BEQ.S   didfind     ;yep, so it passes
            CMP.B   #' ',(A1,D1) ;followed by a space?
            BEQ.S   didfind     ;yes, so we've found it

;this wasn't paydirt, so keep panning
@1          MOVE.W  D2,D0       ;restore chars remaining count
            BMI.S   nofind      ;forget it if we ran out of text
            JSR     bigloop     ;keep looking
            BRA.S   wloop


;this code will find the string if it lies anywhere in the text
findchars   MOVE.L  text(A6),A0 ;point A0 to chars to search
            MOVE.W  count(A6),D0;size of text block
bigloop     MOVE.L  findstr(A6),A1;point A1 to chars to find
            MOVE.W  (A1)+,D1    ;get length byte and 1st char. (skip 'em)
            CMP.W   #255,D1
            BGT.S   @1          ;enter loop if length<>0
            ADDQ.L  #4,A7       ;strip findchar's return address
            BRA     duhn        ;return having done nothing
```

```
;search for first character
@0          CMP.B    (A0)+,D1     ;this one match 1st character?
@1          DBEQ     D0,@0        ;branch until found or done 'em all
            BNE.S    cnofind      ;skip out if no match on 1st character

            MOVE.B   -2(A1),D1    ;length of findstr
            EXT.W    D1
            SUBQ.W   #1,D1        ;length sans 1st character
            BEQ.S    cfound       ;if Length(findstr)=1, we're done
            CMP.W    D1,D0
            BLT.S    cnofind      ;fail if findstr is longer than text left
            MOVE.L   A0,D2        ;save this character position
            CMP.W    D1,D1        ;force EQuality
            BRA.S    @3           ;enter loop

@2          CMP.B    (A0)+,(A1)+ ;match so far?
@3          DBNE     D1,@2        ;check until mismatch or end of findstr

            MOVEA.L  D2,A0        ;restore position (cc's unaffected)
            BNE.S    bigloop      ;if no match then keep looking

cfound      MOVEQ    #1,D1        ;return TRUE
                     RTS

cnofind     SUB.W    D1,D1        ;return FALSE
                     RTS


            STRING   PASCAL
oopstring   DC.B     'Pattern not found.'

            END


#additions to the resource file

resource 'DLOG' (129, "Find dialog") {
        {72, 64, 164, 428},
        dBoxProc,
        visible,
        noGoAway,
        0x0,
        129,
        "Find"
};

resource 'DLOG' (130, "Info") {
        {66, 102, 224, 400},
        dboxproc, visible, nogoaway, 0x0, 130, ""
};
```

```
resource 'DITL' (130) {
        {
/* 1 */ {130, 205, 150, 284},
          button {
                  enabled,
                  "OK already"
          };
/* 2 */ {8, 32, 120, 296},
          /* info */
          statictext {
                  disabled,
                  ""
          }
        }
};

resource 'DITL' (129) {
        { /* array DITLarray: 4 elements */
          /* [1] */
          {64, 48, 84, 121},
          Button {
                  enabled,
                  "OK"
          };
          /* [2] */
          {64, 231, 84, 304},
          Button {
                  enabled,
                  "Cancel"
          };
          /* [3] */
          {8, 8, 24, 352},
          StaticText {
                  disabled,
                  "Find what?"
          };
          /* [4] */
          {32, 8, 48, 352},
          EditText {
                  disabled,
                  ""
          }
        }
};

resource 'MENU' (131, "Custom", preload) {
        131, textMenuProc, 0x3, enabled, "Custom",
        {
          "Find Chars…",
                  noicon, "F", nomark, plain;
          "Find Word…",
                  noicon, "W", nomark, plain
        }
};
```

```
type 'CTST' as 'STR ';

resource 'CTST' (0) {
        "Custom Application - Version 1.0"
};

include "CustomPack.code";


#  This makefile puts the program together incl. the CUST pack.

CustomTest          ƒƒ  CustomCalling.a.o CustomTest.p.o ErrSignal.a.o
# the predefined rule for assembly will build CustomCalling.a.o,
#  CustomPack.code
        Link CustomTest.p.o CustomCalling.a.o ErrSignal.a.o ∂
           "{Libraries}"Interface.o ∂
           "{Libraries}"Runtime.o ∂
           "{PLibraries}"Paslib.o ∂
           -o CustomTest
CustomPack.code     ƒ           CustomPack.a.o
        Link CustomPack.a.o -rt CUST=69 -o CustomPack.code
# Put the resource file together (including the custom code resource)
CustomTest          ƒƒ          CustomTest.r CustomPack.code
        Rez CustomTest.r -a -o CustomTest
```

#136: Register A5 Within GrowZone Functions

See also: The Memory Manager
Technical Note #25—Register A5 Within Trap Patches

Written by: Chris Derossi July 1, 1987
Updated: March 1, 1988

If you have a grow zone function, it may get called when a system routine is trying to allocate memory. Because this can happen, you can't be guaranteed that register `A5` will be correct.

If your grow zone function depends on `A5`, you should save register `A5`, load `A5` from the low-memory global `CurrentA5` (a long word at `$904`), and restore the caller's `A5` before you exit.

From high-level languages, you can also use the Operating System Utility calls `SetUpA5` and `RestoreA5` (page 386 of *Inside Macintosh Volume II*). `SetUpA5` stores the 'old' `A5` on the stack and puts the value stored at `CurrentA5` into `A5`. Make sure to call `RestoreA5` when you're done so that it can pop the saved value of `A5` off the stack.

Your grow zone function depends on `A5` if it does any of the following:

- Accesses your application's global variables (which are stored at negative offsets from `A5`).
- Accesses the QuickDraw globals. (`A5` contains the address of a pointer to the QuickDraw global variables.)
- Makes any ROM trap calls.
- Makes any intersegment calls to routines in your application.

To do any of these, `A5` needs to contain the value from `CurrentA5`. Please note that this is different than the method for calling the ROM from trap patches, where `A5` should retain the value it had upon entry to your patch.

#137: AppleShare 1.1 Server FPMove Bug

See also: *AppleTalk Filing Protocol*

Written by: Rich Andrews
Modified by: Bryan Stearns
Updated:

June 16, 1987
July 1, 1987
March 1, 1988

A bug has been discovered in AppleShare 1.1's implementation of the AppleTalk Filing Protocol `FPMove` call. This bug **only** affects developers implementing custom workstation access code that will access AppleShare 1.1 servers from non-Macintosh systems (such as MS-DOS systems); if the guidelines below are not followed, data loss may result.

The AppleShare file server supports an AFP call known as `FPMove`, used to move a file or directory tree from one place to another on an AppleShare volume. In addition to moving, the caller can specify a new name for the file or directory being moved; in essence, a move and a rename can be accomplished by a single call.

The AppleShare 1.1 server implements this call as follows: the file is moved from the source directory to an invisible holding directory, renamed, then moved to the destination directory. The problem occurs when a locked file is moved and renamed in this manner: the initial move succeeds, the rename fails, and the file is left in the holding directory (essentially lost, as it will be deleted when the server is shut down).

Macintosh AppleShare 1.1 workstation software never uses the move-and-rename combination, so this problem cannot occur on a Macintosh; however, if you're implementing your own workstation-access software for some other machine or operating system, and wish to use this feature, you must follow this procedure:

When a move and rename call comes from the native file system, issue an `FPGetFileDirParms` call to see if the object is a locked file. If it is, issue an `FPSetFileParms` call to unlock the file. Then issue the `FPMove` call, followed by another `FPSetFileParms` call to lock the file again.

AFP does not allow locked files to be renamed, whereas some native file systems (such as MS-DOS) do. You must therefore preflight for this condition to maintain transparency.

This problem will be corrected in a future version of the AppleShare server software.

#138: Using KanjiTalk with a non-Japanese Macintosh Plus

See also: *KanjiTalk Usage Notes*
Script Manager Developers Package

Written by: Priscilla Oppenheimer July 1, 1987
Updated: March 1, 1988

This Technical Note describes the minor differences between using KanjiTalk with the Japanese Macintosh Plus and KanjiTalk with a standard Macintosh Plus.

There are two differences between the Japanese Macintosh Plus and the standard Macintosh Plus: The Japanese Macintosh Plus has the Kanji 12 and 18 point fonts in ROM and it is shipped with the Kana keyboard. It is not necessary to have this keyboard in order to use KanjiTalk. (See the KanjiTalk Usage Notes for details on how to use it with a non-Kana keyboard.) It is, however, necessary to have 12 point Kanji in order to use KanjiTalk; the 18 point Kanji is optional.

When using KanjiTalk with a standard (non-Japanese) Macintosh, the user supplies these fonts on disk and the Macintosh loads them into RAM. At boot time, the Macintosh looks for the 12 point Kanji font file in the system folder of the boot disk. If it cannot find the font, it will look through the root directory of all mounted volumes. (The font has to be at the root level; it cannot be in a folder.) If it still doesn't find the font, it will prompt the user to insert a disk with the font file in the root directory. Once KanjiTalk finds the 12 point font, it will go through the same process looking for the 18 point font. The user can cancel this search if the optional 18 point font is not necessary.

When KanjiTalk finds the fonts, it loads them into memory. The 12 point font takes up approximately 100K of memory and the optional 18 point font takes up approximately 250K of memory. The KanjiTalk code itself takes up about 180K of memory. Because the fonts take up quite a bit of memory, many applications will not work on a Macintosh 512K with the Kanji fonts installed.

Accessing the fonts from ROM is faster, but we have not noticed any significant speed problems when the fonts are accessed from RAM. There is, however, a noticeable difference in speed when the Macintosh is booted. It takes a couple of seconds to load the 12 point font and about 6 seconds to load the 18 point font.

Note that the Japanese Macintosh is unique; Apple has not produced other foreign versions of the Macintosh for different scripts. The introduction of the Arabic Interface System, for example, did not include an Arabic ROM version.

#139: Macintosh Plus ROM Versions

Written by: Cameron Birse July 1, 1987
Updated: March 1, 1988

Readers Digest condensed version of Macintosh Plus ROM history, or the truth according to Bo3bdar the everpresent:

1st version (Lonely Hearts, checksum `4D 1E EE E1`):

Bug in the SCSI driver; won't boot if external drive is turned off. We only produced about one and a half months worth of these.

2nd version (Lonely Heifers, checksum `4D 1E EA E1`):

Fixed boot bug. This version is the vast majority of beige Macintosh Pluses.

3rd version (Loud Harmonicas, checksum `4D 1F 81 72`):

Fixed bug for drives that return Unit Attention on power up or reset. Basically took the SCSI bus Reset command out of the boot sequence loop, so it will only reset once during boot sequence. This version shipped with the platinum Macintosh Pluses.

And Bo3bdar saith: "Thou shalt not rev them damn ROMs no more!"

Later that same day...

Bo3bdar Saith Also:

> Lonely Heifer was about a 2 byte change,
> Loud Harmonica was about 30 byte change.
> No other bug fixes in SCSI or elsewhere.
> Modified object code directly.
> Not possible to get a specific ROM since they are all the same part number.
> Shouldn't rely on a specific ROM, there will be no upgrade.
> Bo3b Bo3b a boola, a wiff Ba2m Bo1om.

#140: Why PBHSetVol is Dangerous

See also: The File Manager

Written by: Chris Derossi July 1, 1987
Updated: March 1, 1988

This note explains `PBHSetVol`, and why its use is not recommended.

`PBHSetVol`, like `SetVol` and `PBSetVol`, allows you to set the current default volume and directory to be used with subsequent File Manager calls. Unlike `SetVol` and `PBSetVol`, though, `PBHSetVol` lets you specify the volume and the directory separately, using the `ioVRefNum` and `ioWDDirID` fields.

`PBHSetVol` lets you specify a `WDRefNum` for the `ioVRefNum` in addition to a partial pathname in `ioNamePtr`. `PBHSetVol` will start at the specified working directory and use the partial pathname to determine the final directory. This directory might not correspond to an already existing working directory, so the File Manager cannot refer to this directory with a `WDRefNum`. Instead it must use the actual volume `refNum` and the `dirID` number (which is assigned when the directory is created, and doesn't change).

The net effect of all of this is, if you call `PBHSetVol`, the File Manager stores the actual volume `RefNum` as the default volume, and the default `DirID` separately. This happens on all calls to `PBHSetVol`. Subsequent calls to `GetVol` or `PBGetVol` will return only the volume `RefNum` in the `ioVRefNum` field of the parameter block. If any code tries to use the `RefNum` returned by `GetVol`, it will be accessing the root of the volume, and not the current default directory as expected.

This is particularly nasty for desk accessories because they don't know that your code has called `PBHSetVol` and they don't get what they expect if they call `GetVol`.

It is therefore recommended that you avoid using `PBHSetVol` because of this side effect. None of the other '`H`' calls that allow you to specify a `DirID` do this, so they're still OK.

#141: Maximum Number of Resources in a File

See also: The Resource Manager

Written by: Cameron Birse July 1, 1987
Updated: March 1, 1988

This note describes the limitation of the number of resources in a single resource file.

There is a limit to the number of the resources in a single resource file. This limitation is imposed by the resource map. There are two bytes at the end of the resource map which are the offset from the beginning of the resource map to the beginning of the resource names list. If there is only one type of resource, then the overhead, from the beginning of the resource map to the beginning of the reference list, is 38 bytes. Since the offset is a two byte value, and is a **signed** number, its highest possible value is 32767. This is the limitation. If you subtract 38 bytes for the overhead, and divide the difference by 12 (the number of bytes for each reference) you get about 2727.4—the limit to the number of resources in a single file is 2727.

The Resource Manager was not intended to manage large numbers of resources, and as a result, its performance is particularly bad with many resources. Because of these restrictions, we recommend that developers avoid using the Resource Manager as a data base tool.

#142: Avoid Use of Network Events

See also: AppleTalk Manager

Written by: Bryan Stearns
Updated:

July 1, 1987
March 1, 1988

Future System software enhancements will not support network events. This note gives hints on weaning your application from the use of network events.

What are network events?

When the Event Manager was designed, an event number was reserved for future support of "network events". Later, when the AppleTalk Pascal Interfaces were written, a completion routine was created that, when an asynchronous AppleTalk operation finished, would post an event using `networkEvt` in the `evtNum` field.

Only the AppleTalk Pascal Interfaces generate network events. Assembly-language users of the AppleTalk drivers (and those who called the AppleTalk drivers directly from high-level languages, using `PBControl` calls) either provide a completion routine of their own, or poll the `ioResult` field of the parameter block passed with the call (when `ioResult` became negative or zero, the call is complete).

Why not use network events?

In some cases, network events can be lost. If the Event Manager finds that the queue is full while posting an event, it discards the oldest event. In a situation (such as a server) where multiple asynchronous ATP requests may complete at once, there is a chance that events may be dropped off the end of the queue. This is more likely if the same machine is also handling user-interface events (like keypresses and mouse actions).

Also, in developing improvements to our operating system, it has become apparent that to continue support of network events, we would have to compromise future enhancements to our system. So, future versions of the Macintosh operating system may ignore network events instead of passing them to the application.

How can I tell that my calls have completed without using network events?

As described on page II-275 of *Inside Macintosh*, you can poll the `abResult` field of the call's `ABusRecord`; when this value becomes negative or zero, the call has completed. You can do this in your main event loop.

With this technique, you can ignore any network events returned by `GetNextEvent`, since the AppleTalk Pascal Interfaces will be posting events anyway. If your application starts enough asynchronous operations, it's possible that their network events will cause other non-network events to be lost. To prevent this, you should call `FlushEvents(networkMask,0)` frequently to purge any accumulated network events from the event queue.

You may also consider using the new preferred high-level interface calls; see Technical Note #132 for more information.

#143: Don't Call ADBReInit on the SE with System 4.1

See also: The Apple Desktop Bus

Written by: Mark Baumwell July 1, 1987
Updated: March 1, 1988

Because of a bug (which causes auto-repeat) in the ROM version of the Macintosh SE keyboard driver, a patch was placed in System 4.1. If `ADBReInit` is called, the ROM version of the keyboard driver will be reloaded, and the RAM version of the driver with the patches will not be used. Therefore, it is recommended that `ADBReInit` not be called on the Macintosh SE until the problem is fixed. (There is no need to call `ADBReInit`.) This problem will not occur with the Macintosh II ROM version of the keyboard driver.



#144: Macintosh Color Monitor Connections

Revised by: Jim Luther & Wayne Correia
Written by: Mark Baumwell

February 1991
July 1987

This Technical Note describes how to connect the Macintosh II Video Card, Macintosh IIci built-in video, and Macintosh LC video to third-party monitors.
Changes since February 1990: Added pinout description for the Macintosh LC external video connector and a Macintosh LC to VGA monitor adapter cable. Standardized signal names throughout Note.

Table 1 documents the pinout descriptions of the Macintosh II Video Cards and the Macintosh IIci built-in video:

Pin Number	Signal Name	Signal Description
1	RED.GND	Red ground
2	RED.VID	Red video signal
3	/CSYNC	Composite synchronization signal
4	SENSE0	Monitor sense signal 0
5	GRN.VID	Green video signal (with sync)
6	GRN.GND	Green ground
7	SENSE1	Monitor sense signal 1
8	n.c.	Not connected
9	BLU.VID	Blue video signal
10	SENSE2	Monitor sense signal 2
11	C&VSYNC.GND	Ground for CSYNC and VSYNC
12	/VSYNC	Vertical synchronization signal
13	BLU.GND	Blue ground
14	HSYNC.GND	HSYNC ground
15	/HSYNC	Horizontal synchronization signal
Shell	CHASSIS.GND	Chassis ground

A slash (/) at the beginning of a signal name indicates that the signal is active low.

Table 1–Macintosh II Video Card and Macintosh IIci Built-in Video

Note: The Macintosh II High-Resolution Display Video Card is the newer replacement for the original four- and eight-bit Macintosh II Video Card (M0211 and M5640). This new card is sold in four- and eight-bit configurations (M0322 and M0324, respectively). The external video connector on the early version of the Macintosh II Video Card did not have the signals SENSE0, SENSE1, and SENSE2.

Note: The newer Macintosh II Video Cards and Macintosh IIci built-in video require that pin 4 (SENSE0) be connected to Ground to signal the connection of a 640 x 480 monitor. Do not connect pins 7 or 10 as they are unused on original Macintosh II

Video Cards and there are built-in pullup resistors on the newer Macintosh II Video Card and Macintosh IIci to terminate these pins when not in use.

Table 2 documents the pinout descriptions of the Macintosh LC video connector:

Pin Number	Signal Name	Signal Description
1	RED.GND	Red ground
2	RED.VID	Red video signal
3,15	/CSYNC or /HSYNC	Composite synchronization signal if Apple monitor. Horizontal synchronization signal if VGA monitor.
4	SENSE0	Monitor sense signal 0
5	GRN.VID	Green video signal
6	GRN.GND	Green ground
7	SENSE1	Monitor sense signal 1 (grounded internally)
8	n.c.	Not connected
9	BLU.VID	Blue video signal
10	SENSE2	Monitor sense signal 2
11	C&VSYNC.GND	Ground for CSYNC and VSYNC
12	/VSYNC	Vertical synchronization signal
13	BLU.GND	Blue ground
14	HSYNC.GND	HSYNC ground
Shell	CHASSIS.GND	Chassis ground

A slash (/) at the beginning of a signal name indicates that the signal is active low.

Table 2–Macintosh LC External Video Connector

Note: The Macintosh LC does not supply vertical synchronization with the Green video signal (pin 5). The vertical synchronization signal is supplied on pin 12.

Note: The Macintosh LC requires that pin 4 (SENSE0) be connected to Ground to signal the connection of a 640 x 480 monitor. The Macintosh LC requires that pin 4 and 10 (SENSE0 and SENSE2) be connected to Ground to signal the connection of a 512 x 384 monitor (i.e., the Macintosh 12" RGB Display). The Macintosh LC requires that pin 10 (SENSE2) be connected to Ground to signal the connection of a VGA monitor. Pin 7 (SENSE1) is grounded in the Macintosh LC.

Macintosh II to Sony Multiscan (CPD-1302)

To connect a Macintosh II to a Sony Multiscan monitor, you need to make an adapter cable from the video card to the monitor (which has a 9-pin D-type connector). Following is the pinout description for the adapter cable (using the automatic sync-on-green configuration):

Macintosh II Video Card Pin	Sony Pin	Signal Name
1	1	Ground
2	3	Red video signal
4	1	SENSE0 Grounded
5	4	Green video signal (with sync)
9	5	Blue video signal

Macintosh II to NEC MultiSync (JC-140IP3A)

To connect a Macintosh II to a NEC MultiSync monitor, you need to make an adapter cable from the video card to the monitor (which has a 9-pin D-type connector). Following is the pinout description for the adapter cable (using the automatic sync-on-green configuration):

Macintosh II Video Card Pin	NEC Pin	Signal Name
1	6,7,8,9	Ground
2	1	Red video signal
4	6,7,8,9	SENSE0 Grounded
5	2	Green video signal (with sync)
9	3	Blue video signal

The monitor must be set to Analog mode and Manual mode. This adaptor cable also works with an equivalent monitor such as the Taxan Super Vision 770.

Macintosh LC to VGA

The Macintosh LC can supply a 640 x 480, VGA timed signal for use with VGA monitors by using an adapter cable. The standard Macintosh LC supports VGA to 16 colors, and with the optional 512K VRAM SIMM, the VGA monitor is supported to 256 colors.

Note: The Macintosh LC supplies signals capable of driving TTL level inputs. However, some low impedance input VGA monitors do not work with the Macintosh LC.

To connect a Macintosh LC to a VGA monitor, you need to make an adapter cable from the Macintosh LC video connector to the VGA monitor. Following is the pinout description for the adapter cable:

Macintosh LC Video Connector	VGA Pin	Signal Name
1	6	Red ground
2	1	Red video signal
5	2	Green video signal
6	7	Green ground
9	3	Blue video signal
13	8	Blue ground
15	13	/HSYNC
12	14	/VSYNC
14	10	HSYNC ground
7,10	nc	SENSE1 & SENSE2 tied together

VGA monitors are identified by shorting pin 7 to pin 10 on the Macintosh LC video connector. The Macintosh LC grounds pin 7 on its video connector, which results in pulling down pin 10 and gives the correct monitor ID for a VGA monitor.

Further Reference:

- *Guide to the Macintosh Family Hardware*, Second Edition
- *develop*, "Macintosh Display Card 8•24 GC: The Naked Truth," July 1990



#145: Debugger FKEY

Revised by: Mark Baumwell
Written by: Mark Baumwell

August 1990
July 1987

This Technical Note formerly discussed showed how to write an `'FKEY'` to trap to the debugger.
Changes since March 1988: Merged the contents of this Note into Technical Note #256, Stand-Alone Code, *ad nauseam.*

This Note formerly showed how to write an `'FKEY'` resource to trap to the debugger. This information is now an example of writing stand-alone code resources in Technical Note #256, Stand-Alone Code, *ad nauseam.*

Macintosh
Technical Notes

Developer Technical Support

#146: Notes on MPW's -mc68881 Option

Revised by: Dave Radcliffe May 1992
Written by: Bryan Stearns July 1987

This Technical Note discusses MPW's `-mc68881` option, which represents `Extended` values in 96 bits (instead of 80, as with software SANE), and compatibility issues when using non-SANE system calls that expect 80-bit `Extended` values.

Changes since June 1990: Extended the warning about explicitly checking for the presence of an FPU if an application uses floating-point instructions to include the possibility of FPU-less MC68040 products and also raised the issue of extended values embedded in data structures.

MPW Compilers and Extended Values

The MPW 2.0 and later compilers provide a command-line option, `-mc68881`, to generate in-line code to use the Motorola 68881/68882 floating-point units (FPU). (Note that MPW compilers currently do not include a `-mc68882` option, as they treat the two chips the same. If you want to optimize code for the 68882, then you need to write your own assembly-language code.) This option allows applications to sacrifice compatibility with other Macintosh models (those not equipped with an FPU) in exchange for much increased numeric performance.

Warning: Applications should not make assumptions about the presence of an FPU based upon the microprocessor of a Macintosh. If an application makes a conditional branch to execute floating-point instructions directly, then it should first explicitly check for the presence of the FPU with `_Gestalt` or `_SysEnvirons`. Furthermore, you should not assume that the presence of an MC68040 processor guarantees access to floating-point instructions. Motorola has announced a reduced-cost MC68040 CPU that lacks an integral floating-point unit. Possible inclusion of such a CPU in future Macintosh products means FPU-less 68040 products may someday exist.

When using the `-mc68881` option, the compiler stores all `Extended` values in the 96-bit format used by the 68881 instead of the 80-bit software SANE (Standard Apple Numerics Environment), both of which are illustrated in Figures 1 and 2.



Figure 1 Software SANE Format (80-Bit)

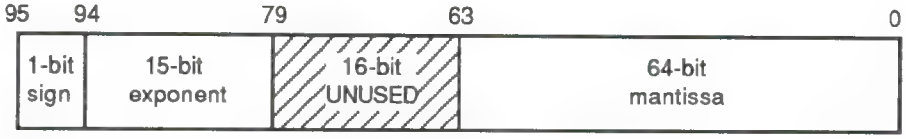


Figure 2 MC68881 Format (96-Bit)

This difference in format affects all procedures that accept floating-point values as arguments, since all floating-point arguments are converted to `Extended` before being passed, no matter how they are declared (for example, `Real`, `Single`, `Double`, or `Comp`).

When compiling with this option, you must link with a special SANELib library file, SANE881Lib.o; the interface source file SANE.p contains conditional-compilation statements to make sure that the correct library's interface is compiled. In this situation, SANE procedures are used for certain transcendental functions only (see the last section of this Tech Note), and these functions, which are in SANE881Lib.o, expect their `Extended` parameters in 96-bit format.

However, numeric routines that are not compiled by Pascal (such as any assembly-language routines) have no way of finding out that their parameters are in 96-bit format. If it is not possible to rewrite these routines for 96-bit values, you can use the SANELib routines `X96ToX80` and `X80ToX96` to convert between formats. It might be simplest to define a new interface routine that automatically converts the formats:

Pascal

```
{FPFunc is a generic floating-point, assembly-language function that accepts}
{an 80-bit Extended parameter and returns an 80-bit result.}
{We've changed the types to reflect that these are not 96-bit values.}

FUNCTION FPFunc(x: Extended80): Extended80; EXTERNAL;

{Given that we're compiling in -mc68881 mode, the compiler}
{thinks that Extended values are 96-bits long, but FPFunc wants an}
{80-bit parameter and produces an 80-bit result; we convert.}

FUNCTION FPFunc96(x: Extended): Extended; {x is a 96-bit extended!}
BEGIN
  {convert our argument, call the function, then convert the result}
  MyFPFunc := X80ToX96(FPFunc(X96ToX80(x))); {call the real FPFunc}
END;
```

C

```
extern Extended80 FPFunc (Extended80 x);

Extended FPFunc96 (Extended x); //x is a 96-bit extended!
{
  //convert our argument, call the function, then convert the result
  MyFPFunc = X80ToX96(FPFunc(X96ToX80(x))); //call the real FPFunc
}
```

It's best to avoid compiling some parts of an application with the `-mc68881` option on and other parts with it off; very strange bugs can occur if you try this. Note that 80-bit code and 96-bit code cannot reference the same `Extended` variables. There is no way to tell whether a given stored value is in 80-bit format or 96-bit format.

Size of Data Structures

Compile time differences in the size of extended variables also cause problems for variables embedded in data structures. The size of data structures and offsets within the structure can change depending on the compile-time options. This could be particular troublesome of you write the data to a file and then attempt to read the file with a version of the program which was compiled differently.

An example of this can be found in some of the Sound Manager interfaces. For example, prior to MPW 3.2.1, the ExtSoundHeader record contained an extended field which could cause the problem described above. In MPW 3.2.1, the extended field was changed to the uniform type extended80 to avoid the problem.

SANE on the Macintosh II

The version of SANE provided in the Macintosh II ROM recognizes the presence of the 68881 and uses it for most calculations automatically. SANE still expects (and produces) 80-bit-format `Extended` values; it converts to and from 96-bit format internally when using the 68881.

A Note About 68881 Accuracy and Numeric Compatibility

SANE is more accurate than the 68881 when calculating results of certain functions (`Sin`, `Cos`, `Arctan`, `Exp`, `Ln`, `Tan`, `Exp1`, `Exp2`, `Ln1`, and `Log2`). To maintain this accuracy, SANE does not use 68881 instructions to directly perform these functions; thus the results you get from SANE calculations are still identical on all Macintosh systems.

To preserve this numeric compatibility with other SANE implementations, MPW compilers normally do not generate in-line 68881 calls to the above functions, even when the `-mc68881` option is used; instead, they generate SANE calls to accomplish them. If you are willing to sacrifice numeric compatibility to gain extra speed, you can override this compiler feature with the compile-time variable, `Elems881`; include the option `-d Elems881 = TRUE` on the Pascal compiler and `-Elems881` on the C compiler command line to cause the compiler to generate direct 68881 instructions.

For certain other transcendental functions provided by the 68881 that are not provided by SANE, MPW compilers generate direct 68881 calls if the `-mc68881` option is on, independent of the setting of the `Elems881` variable. These operations are `Arctanh`, `Cosh`, `Sinh`, `Tanh`, `Log10`, `Exp10`, `Arccos`, `Arcsin`, and `Sincos`.

For Pascal programmers, it is important to note that if you want an application to check for an FPU and exit gracefully if it does not exist, then you need to check for the FPU with code that does not have the `-mc68881` option turned on. You need to do this because the `-mc68881` option inserts code to initialize the 68881/68882 at the beginning of your code, and this initialization code causes an exception error if no FPU is present. For example, if you check for the existence of an FPU in your main Pascal procedure, you need to compile that main procedure with `{$MC68881-}`. Note that this compiler option affects the entire file that contains the option, so you would need to separate any code that uses an FPU into another file.

After you determine that an FPU exists, you have to execute the following instructions by hand to initialize the FPU yourself:

```
PROCEDURE ClearTheFPU ();
      INLINE    $42A7,                  {clr.l -(a7)}
                $42A7,                  {clr.l -(a7)}
                $F21F, $9800;           {fmovem (a7)+,FPCR/FPSR}
```

Further Reference:

- *Inside Macintosh*, Volume V-1, Compatibility Guidelines
- *Apple Numerics Manual*, Second Edition
- Macintosh Technical Note #129, _Gestalt & _SysEnvirons—a Never-Ending Story
- Macintosh Technical Note #236, Speedy the Math Coprocessor
- MPW Pascal Reference Manual

#147: Finder Notes: "Get Info" Default & Icon Masks

See also: Technical Note #48—Bundles

Written by: Bryan Stearns July 1, 1987
Updated: March 1, 1988

The Finder has undergone a couple of changes you should keep in mind when creating the "bundle" information for your application.

Creator String will be the default "Get Info" comment text

The "creator" (or "signature") string (contained in a resource whose type is your application's four-character creator type, and whose ID is 0) will be used as the default for the comment text displayed by the Finder's "Get Info" command. Thus, you should set up this string (when you build your application) to contain the name of your program and a version number and date.

Icon Masks should match their icons

Your application's `BNDL` ("bundle") resource ties the file types that it uses for its documents with the icons to be displayed for those documents. For each icon, a "mask" icon is also provided; this mask is used to punch a hole in the gray desktop before drawing the icon.

Some applications use a cleverly-modified mask to provide an "action icon" that looks different when it's selected. This causes problems; it is important that the mask be what it's supposed to be (a solid black copy of the icon).

#148: Suppliers for Macintosh II Board Developers

See also: *Designing Cards and Drivers for the Macintosh II and Macintosh SE*

Written by: Mark Baumwell July 1, 1987
Updated: March 1, 1988

This note lists suppliers of parts that may be helpful for Macintosh II board developers. If your company supplies these parts, but is not listed here, please send a message to us (at the address on Technical Note #0) and we'll include you in the next revision of this technical note.

This is a list of companies that supply the Macintosh II expansion port cover (p/n 805-5064-05) (Foldout 2 in *Designing Cards and Drivers or the Macintosh II and Macintosh SE*). It is not intended to be an endorsement or an indication of quality; it is just our list of known suppliers.

Galgon Industries, Inc.
37399 Centralmont Place
Fremont, CA 94536
Attn: Ron Haddox—General Sales
(415) 792-8211

Vector Electronics
12460 Gladstone Ave
Sylmar, CA 91342
(818) 365-9661
FAX# 818-356-5718
Attn: Norm Brunell

North American Tool and Die
999 Beecher Street
San Leandro, CA 94577
(415) 632-9263
Attn: Glenn Erikson

In addition to supplying the expansion port cover, Vector Electronics supplies Macintosh II NuBus extender boards and prototyping boards.

#149: Document Names and the Printing Manager

See also: The Printing Manager
Technical Note #122—Device-Independent Printing

Written by: Bryan Stearns July 1, 1987
Updated: March 1, 1988

Our compatibility testing for LaserShare (Apple's LaserWriter spooler) has turned up a number of applications that do not provide the Printing Manager with a document name; although this feature is not required, it is nice for users that share printers.

Some printers (usually those that are shared between many users, like the LaserWriter) can provide the names of the users who are printing and the documents that are being printed to others interested in using the printer.

If the chosen printer uses a document name, the Printing Manager gets the name from the frontmost window's title. If there is no front window, or if the window's title is empty, the Printing Manager defaults to "unknown."

This method was chosen because it works most transparently to applications; however, it won't work if your application doesn't display windows when printing (for instance, many applications that use windows for their documents do not open their documents when printing in response to a Finder "Print" command).

As a general solution to this problem, you can put up a window containing a message like "Press ⌘–. to cancel printing", and give it the document's title. If the window is one that doesn't have a title bar (like `dBoxProc`), this title will not be displayed. MacApp takes this approach. If for some reason you don't want to put up a visible window, you can create a tiny window and hide it behind the menu bar: for instance, global coordinates of (1,1,2,2). Make sure you use a `plainDBox`, so that no title will be drawn (otherwise, in the unlikely case that a user is using a Macintosh II with two stacked screens, main screen on the bottom, the title might be visible on the upper screen).

Since the Printing Manager checks the name at `PrValidate` time, call `PrValidate` after `PrCloseDoc` and before the next `PrOpenDoc`, if you want unique names.

A number of applications set the document name in the print record directly. You should not do this because a) not all printers support this field, and b) none are guaranteed to support it in the future. (Apple does not guarantee that internal fields of the Printing Manager's data structures will remain the same; the Printing Manager is targeted for substantial internal change!)

#150: Macintosh SE Disk Driver Bug

Written by: Mark Baumwell
Updated:

July 1, 1987
March 1, 1988

A bug in the Macintosh SE ROMs causes the top drive to be slower than the bottom one in two-drive machines. This bug is fixed in System 4.2 and newer.

#151: System Error 33, "zcbFree has gone negative"

See also: The Memory Manager

Written by: Bryan Stearns July 1, 1987
Updated: March 1, 1988

System 3.2 introduced a new system error, ID=33, generated by the Memory Manager when it notices that a heap had been corrupted in a certain way. This error is listed in the file "SysErr.a" as "`negZcbFreeErr`".

The Memory Manager will trigger an "ID=33" system error when, during some operation which scans the objects in the heap, it sees that its running count of free bytes (`zcbFree`, an internal value) has become negative (an impossible feat in normal operation). This is nearly always caused by writing zeros past the end of one of the Memory Manager's heap blocks (overwriting and corrupting the next block's header, making it appear to be a free block).

If you get this error, use a debugger (like Macsbug or TMON) when you attempt to reproduce the error, to check the consistency of the heap up to the point where the error occurs. You may need to do this repeatedly until you isolate the operation that is corrupting the heap.

Note that although the heap may become corrupted during a system call, this doesn't mean you've found a bug in the ROM; your code could be passing incorrect or invalid parameters to this or a previous system call, or could have corrupted a data structure used by a system call. More debugging is usually in order in this case; tools like Discipline (included in TMON; also available from users' groups and electronic services) can help detect invalid parameters in system calls. Also, there is a Macsbug command, AH, that can check the consistency of the heap on every system call. See the documentation that came with your debugger to see what special features it offers.

A note about "SysErr.a"

Technical Support is often asked for an up-to-date list of error codes. In general, this is provided in "SysErr.a", the file of error numbers shipped with the most current version of MPW. Admittedly, the documentation value of "SysErr.a" is sometimes low (as in the case of `negZCBFreeErr`), but it may give you a clue as to what the error might mean.

#152: Using Laser Prep Routines

See also: The Printing Manager
PostScript Language Reference Manual
Technical Note #122—Device-Independent Printing

Written by: Ginger Jernigan July 1, 1987
Updated: March 1, 1988

This technical note addresses the issues involved in depending on the procedures and constants defined in the Laser Prep dictionary.

When you are writing an application that uses PostScript heavily, it is very tempting to call the procedures already defined in the Laser Prep dictionary, rather than taking up the space in the printer's memory with PostScript procedures of your own. Or, if a procedure in the dictionary doesn't do what you need it to do, it is tempting to go in and change it to do what you really want.

Unfortunately, we cannot guarantee that either the name or the function of a particular procedure in the dictionary will stay the same when the LaserWriter driver changes. In addition, some procedures may become obsolete and go away when the driver changes. Since the Laser Prep dictionary is considered part of the source for the LaserWriter driver, Apple reserves the right to make any changes to it that are deemed necessary.

Because we cannot guarantee the permanence of the contents of the Laser Prep dictionary, relying on its contents can pose a significant compatibility problem. If you rely on the procedures defined in the Laser Prep dictionary, your application will have to be revised every time Apple releases a new LaserWriter driver.

If you feel that you absolutely must use or modify procedures in the Laser Prep dictionary, you must **always** check the version that is loaded into the printer before you print. This allows your application to take appropriate action if the version of the dictionary that has been downloaded to the printer isn't one that you know about.

How To Check The Laser Prep Dictionary Version

To determine the version of Laser Prep that the printer may contain, you have to communicate with the printer using the Printer Access Protocol (PAP); you can't just send your query through the LaserWriter driver because there is no way to get an answer back. The object code and documentation for PAP are available from Apple's Licensing department.

To determine whether the dictionary has been downloaded and whether it is the right one, send the following PostScript code to the printer:

```
%!PS-Adobe-1.2 Query
%%Title: Query to establish Laser Prep ProcSet version propriety
%%?BeginProcSetQuery: AppleDict md xx
/md where{
/md get /av get cvi xx eq
{(1)}{(2)}ifelse}
{(0)}ifelse = flush
%%?EndProcSetQuery: unknown
%%EOF
```

`md` is the name of the Apple dictionary and `xx` is the version number you want.

Note: `/av` is a constant in the `md` dictionary which contains the dictionary's version number. This is the **only** object in the dictionary whose name and function are guaranteed not to change.

From the printer you will receive a string. If the string returned begins with '%%', it is a `Status` response. You can ignore it and wait for another string.

If the response is '0', the dictionary hasn't been downloaded yet; you will need to determine how to best handle this situation for your application.

If the response is '1', the printer is loaded with the correct version of the dictionary.

If the response is '2', then the dictionary exists but it isn't the version you need. In this case you need to either let the user know, or proceed in as standard a fashion as possible, without calling or modifying routines contained in the Laser Prep dictionary.

Translating PostScript Files

Some applications interpret the PostScript files that are generated by the LaserWriter driver when the user presses command-F (generates document only) or command-K (generates dictionary and document) after clicking on the OK button in the Job dialog. A typical application might translate these PostScript files into another page description language.

This kind of application requires intimate knowledge of the contents of the dictionary in order to be able to do the translation, because it may have to expand the procedures used to their actual values before it can then translate the PostScript to another language. This poses a significant compatibility problem. Since we cannot guarantee that the contents of the dictionary will not change, these types of applications will have to be revised every time we release a new version of the LaserWriter driver. Also, there is no way to know which version of the LaserWriter driver generated the PostScript file the application is interpreting. You will have to require that a particular version of the LaserWriter driver be used to generate the PostScript files that your application will interpret.

Printer Independence

Applications that are written to take advantage of the routines in the Laser Prep dictionary are, of course, highly device dependent. Being device dependent can drastically reduce your chances of being compatible with future printer-type devices. For a more detailed discussion of this issue, please refer to Technical Note #122.

Macintosh Technical Notes

#153: Changes in International Utilities and Resources

See also: The International Utilities
The Script Manager

Written by: Priscilla Oppenheimer July 1, 1987
Updated: March 1, 1988

The International Utilities package and the international resources have been changed with System file 4.1 to take advantage of the Script Manager.

INTL vs. itl

In the past, there were two `INTL` resources in the System file, `INTL 0` and `INTL 1`, which contained international formatting options. Starting with System 4.1, `itl0` and `itl1` replace `INTL 0` and `INTL 1`. (`INTL 0` and `INTL 1` are still included with the System file so that applications that do a `GetResource` for them will still find them. Only these erroneous applications that use `GetResource` will access these resources now; the System will no longer look at them.) There can now be a set of international resources (`itl`s) for each script that is installed. That is, where once there was one resource type (`INTL`) with IDs 0 and 1 there are now many different resource types (`itl0`, `itl1`, `itl2`, `itlb`, `itlc`, `KCHR`) with distinct resource IDs for the various countries. The U.S. System file uses resource ID 0 for all `itl` resources.

If your existing application calls `IUGetIntl` to get the appropriate international resource, you will have no problems under System 4.1. If, however, you call `GetResource` on `INTL 0` or `INTL 1` and then modify them, and expect that the System will use the modified resource, you will no longer work correctly. If your application has been released in the past with a modified System file to include your own `INTL 0` and/or `INTL 1`, you may want to release your application with a modified `itl0` and/or `itl1`.

The International Sorting Routine

The international sorting routine is used to handle cases where letters are equal in primary ordering but different in secondary ordering (e.g., "ä" and "a"). The routine can also handle cases where one character sorts as if it were two, or two characters would sort as if they were one, or even character reversals.

Prior to System 4.1, the international sorting routine was stored starting with the `localRtn` field of `INTL 1`. As of System 4.1, this routine is now stored in `itl2`. An `INTL 2` resource is also included with the international resources, just for consistency (the System never uses it). It is exactly the same as `itl2`.

Writing your own sorting routine can be dangerous if it is not done correctly. Before attempting to do this, we would highly recommend you check with Apple to determine if we are already doing one for the language of interest. We plan to do many international versions of System file 4.1, and most developers' needs will be met by these. A new policy at Apple will now make it possible to license foreign versions of the System file from Software Licensing. Contact Software Licensing at (408) 973-4667 for more information about this.

Error in APDA draft of *Inside Macintosh Volume V*

There is an error in the APDA draft of the International Utilities Package chapter of *Inside Macintosh Volume V*. It says that there is a flag in the Script Manager globals that overrides the current font and always uses the `INTL 0` and `INTL 1` resources. This is not true. `INTL 0` and `1` have been replaced by `itl0` and `itl1`, and are not used except by applications that explicitly get them by doing a `GetResource`. There is a relevant Script Manager flag called `IntlForce` that is correctly but incompletely explained in the Script Manager chapter. When `IntlForce` is false, the resources used by the International Utilities are determined by the current script. The script that is in use is determined by which font is in use for the port in use. For example, if you switch to the Kyoto font, then you will be using the Japanese Script Interface System (KanjiTalk). If you do this with `IntlForce` set to false, then you will use the resources that are associated with the Japanese Script Interface System.

When `IntlForce` is true, the International Utilities always use the resources that are associated with the system script; this is usually Roman. `IntlForce` is true by default. Actually, the Script Manager initialization routine reads the `itlc` resource to determine what the default is, and Apple plans to release the various international System files with this field set to true in the `itlc` resource. Applications that want to change the value of `IntlForce` can use the Script Manager call `SetEnvirons` with the `smIntlForce` verb. There is a Script Manager call, `IntlScript`, to find out the current value.

Bug in System 4.1 version of `itl1`

There is a bug in the `itl1` resource in System 4.1. The `itl1` resource contains arrays of day and month names, as did the `INTL 1` resource. Their formats are:

```
days ARRAY [1..7] of STRING[15]
months ARRAY[1..12] OF STRING[15]
```

Every day and month is supposed to be coded as a Pascal string with a maximum of 15 characters, and the actual length in the first byte of the string. In System 4.0 and System 4.1, the contents of each string is always a Pascal string of length 15, with the unused characters set to nulls. In other words, the length byte is set to hex `0F`. This will be fixed in a future release of the System file. If it is currently causing your application problems, you can change the lengths with ResEdit, or change them at run time, or use your own `itl1` resource and release your application with an installer script. The recommended approach is to wait for a fixed version of the System file from Apple.

#154: Displaying Large PICT Files

See also: QuickDraw
Technical Note #21—Internal Picture Format
Technical Note #35—DrawPicture Problem
Technical Note #88—Signals

Written by: Rick Blair July 1, 1987
Updated: March 1, 1988

Now that we have scanners and other massive-picture producing types of applications, there is a need to address the problem of how to display a PICT format object that is bigger than a current PICT resource is allowed to be. Note that this technique applies equally well to version 1 and version 2 (word-opcode) pictures as produced by the Macintosh II.

Future Compatibility

Think of the handle returned by a `GetResource('PICT',ID)` as a "handle" in the more general sense of being an abstract "tag"—something that the ROM routines can use to draw the picture with. Don't assume that the entire picture has been read into memory or that you can directly read any bytes beyond the basic `Picture` record structure (`picSize` followed by `picFrame`). Someday we may provide a mechanism for the resource to be disk- instead of memory-based. The QuickDraw bottleneck procedures will know how to get data from and put data into the pictures in any case.

Spooling from a PICT file

In order to display pictures of arbitrary size, your application should be able to import a QuickDraw picture from a file of type PICT. This is the file produced by a "Save as..." from MacDraw with the PICT option selected.

What follows is a small program fragment that demonstrates how to spool in a picture from [the data fork of] a PICT file. The picture can be larger than the historical 32K resource size. See technical note #88 if you are unfamiliar with the `Signal` mechanism. We assume that a `CatchSignals` has been done before `GetandDrawPICTFile` is called.

MPW Pascal Example

```
{the following variable must be at the top level}

VAR
   globalRef    : INTEGER;      {refNum of the file to read from}

{the following procedure must be at the top level}

PROCEDURE GetPICTData(dataPtr: Ptr; byteCount: INTEGER);
{replacement for the QuickDraw bottleneck routine}

   VAR
      err         : OSErr;
      longCount   : LONGINT;

   BEGIN
      longCount := byteCount;
      err := FSRead(globalRef,longCount,dataPtr);
      {can't check for an error because we don't know how to handle it}
   END;

CONST
   abortPICT    = 128;         {error code if DrawPicture aborted}

PROCEDURE GetDrawPICTFile;     {read in a PICT FILE selected by the user}

   VAR
      wher         : Point;     {where to display dialog}
      reply        : SFReply;   {reply record}
      myFileTypes  : SFTypeList; {more Standard FILE goodies}
      numFileTypes: INTEGER;

      savedProcs   : QDProcsPtr;
      myProcs      : QDProcs;   {use CQDProcs for a color window}

      myPicture    : PicHandle; {we need a picture handle for DrawPicture}
      longCount    : LONGINT;
      myEOF        : LONGINT;
      myFilePos    : LONGINT;

   BEGIN
      wher.h := 20;
      wher.v := 20;
      numFileTypes := 1;        {display PICT files}
      myFileTypes[0] := 'PICT';
      SFGetFile(wher,'',NIL,numFileTypes,myFileTypes,NIL,reply);

      IF reply.good THEN BEGIN
         SetStdProcs(myProcs); {use SetStdCProcs for a CGrafPort}
         myProcs.getPicProc := @GetPICTData;
         savedProcs := thePort^.grafProcs; {set the grafProcs to ours}
         thePort^.grafProcs := @myProcs;

         myPicture := PicHandle(NewHandle(SizeOf(myPicture)));

         Signal(FSOpen(reply.fname,reply.vRefNum,globalRef));
         Signal(GetEOF(globalRef,myEOF)); {get EOF for later check}
```

```
        Signal(SetFPos(globalRef,fsFromStart,512)); {skip header}

        {read in the (obsolete) size word and the picFrame}
        longCount := SizeOf(myPicture);
        Signal(FSRead(globalRef,longCount,Ptr(myPicture^)));

        DrawPicture(myPicture,myPicture^^.picFrame); {draw the picture}

        Signal(GetFPos(globalRef,filePos)); {get position for check}
        Signal(FSClose(globalRef));

        DisposHandle(Handle(myPicture));

        thePort^.grafProcs := savedProcs; {restore the procs}

        {Check for errors. If there wasn't enough room,}
        {DrawPicture will abort; the FILE position mark}
        {won't be at the end of the FILE.}
        IF filePos <> myEOF THEN Signal(abortPICT);
      END; {IF reply.good}
    END; {GetDrawPICTFile}
```

MPW C Example

```
/*replacement for the QuickDraw bottleneck routine*/
pascal void GetPICTData(dataPtr,byteCount)
Ptr             dataPtr;
short           byteCount;


{ /* GetPICTData */
     OSErr          err;
     long           longCount;

     longCount = byteCount;
     err = FSRead(globalRef,&longCount,dataPtr);
     /*can't check for an error because we don't know how to handle it*/
} /* GetPICTData */

/*error code if DrawPicture aborted*/
#define      abortPICT     128

OSErr GetDrawPICTFile()       /*read in a PICT FILE selected by the user*/

{    /* GetDrawPICTFile */

          Point          wher;       /*where to display dialog*/
          SFReply        reply;      /*reply record*/
          SFTypeList     myFileTypes;/*more Standard FILE goodies*/
          short          numFileTypes;
          OSErr          err;
          QDProcsPtr     savedProcs;
          QDProcs        myProcs;/*use CQDProcs for a color window*/
          PicHandle      myPicture;
                         /*we need a picture handle for DrawPicture*/
          long           longCount,myEOF,filePos;
```

```
        wher.h = 20;
        wher.v = 20;
        numFileTypes = 1;
              /*display PICT files*/
        myFileTypes[0] = 'PICT';
        SFGetFile(wher,'',nil,numFileTypes,myFileTypes,nil,&reply);

        if (reply.good)
        {
              SetStdProcs(&myProcs);
              /*use SetStdCProcs for a CGrafPort*/
              myProcs.getPicProc = GetPICTData;
              savedProcs = (*qd.thePort).grafProcs;
              /*set the grafProcs to ours*/
              (*qd.thePort).grafProcs = &myProcs;

              myPicture = (PicHandle)NewHandle(sizeof(Picture));

              err = FSOpen(&reply.fName,reply.vRefNum,&globalRef);
              if (err != noErr) return err;

              err = GetEOF(globalRef,&myEOF);
              /*get EOF for later check*/
              if (err != noErr) return err;

              err = SetFPos(globalRef,fsFromStart,512);/*skip header*/
              if (err != noErr) return err;

              /*read in the (obsolete) size word and the picFrame*/
              longCount = sizeof(Picture);
               err = FSRead(globalRef,&longCount,(Ptr)*myPicture);
              if (err != noErr) return err;

              DrawPicture(myPicture,&(**myPicture).picFrame); /*draw
                                          the picture*/

              err = GetFPos(globalRef,&filePos);/*get position for
                                          check*/
              if (err != noErr) return err;
              err = FSClose(globalRef);
              if (err != noErr) return err;

              DisposHandle((Handle)myPicture);

              (*qd.thePort).grafProcs = savedProcs;/*restore the
                                          procs*/

               /*Check for errors. if there wasn't enough room,*/
               /*DrawPicture will abort; the FILE position mark*/
              /*won't be at the end of the FILE.*/

              if (filePos != myEOF)  return abortPICT;
              else return noErr;
        } /*if (reply.good) */
}     /* GetDrawPICTFile */
```

More on Picture Compatibility

Many applications already support PICT resources larger than 32K. The 128K ROMs (and later) allow pictures as large as memory (or spooling) will accommodate. This was made possible by having QuickDraw ignore the size word and simply read the picture until the end-of-picture opcode was reached.

For maximum safety and convenience, let QuickDraw generate and interpret your pictures.

While Apple has provided you with the data formats that allow you to read or write picture data directly, we recommend that you always let DrawPicture or OpenPicture and ClosePicture process the opcodes.

One reason to read a picture directly by scanning the opcodes would be to disassemble it to, for example, extract a Color QuickDraw pixel map to save off in a private data structure. This shouldn't normally be necessary.

If you do look at the picture data be sure and check the version information. You may want to put up an alert in your application that indicates to the user when a picture was created using a later version of the picture format than your application recognizes, letting them know that some elements of the picture cannot be displayed. If the version information indicates a QuickDraw picture version later than the one recognized by your application, your program should skip over the new opcodes and only attempt to parse the ones it knows.

As with reading picture data directly, it is best to use QuickDraw to create data in the PICT format. If you do need to create PICT format data directly, it is essential that you use the latest opcode specifications and that you thoroughly test the data produced on both color and black and white Macintosh machines. Contact Macintosh Developer Technical Support if you are not sure that you have the latest specifications.

Apple does not guarantee that a picture which wasn't produced by QuickDraw will work.

#155: Handles and Pointers—Identity Crisis

See also: QuickDraw
The Memory Manager

Written by: Jim Friedlander
Updated:

September 1, 1987
March 1, 1988

A handle is a handle and a pointer is a pointer. Applications should avoid embedding non-relocatable objects (that the system assumes will never move) in handles.

In order to avoid fragmentation, some applications embed pointers (non-relocatable memory manager objects) in handles, so that the handles can be moved around as needed. This can cause several problems, especially with the Macintosh II, and should be avoided.

For example, use of a handle to store a `GrafPort` can be particularly dangerous. A `GrafPort` must not move between the time that it is opened (`OpenPort`, `NewWindow`, `NewDialog`, etc.) and the time that it is closed (`ClosePort`, `DisposeWindow`, `DisposDialog`, etc.). Color QuickDraw keeps a list of open ports and **pointers** to them, so, if you create a `GrafPort` and it moves while still open, Color QuickDraw will (unknowingly) have a pointer to outer space instead of a pointer to a `GrafPort`. When it needs to use that pointer, it will get hopelessly confused and probably issue a system error to let you know.

As an aside, if you open a port by calling `OpenPort` or `OpenCPort`, you should always close the port by calling `ClosePort` or `CloseCPort` before calling `DisposPtr` on the port or you will orphan handles (`visRgn`, `clipRgn` and more).

#156: Checking for Specific Functionality

See also: Operating System Utilities
Assembly Language
Technical Note #129—SysEnvirons

Written by: Jim Friedlander September 1, 1987
Updated: March 1, 1988

This technical note explains how to check at run time to see if specific functionality, such as the "new" TextEdit, is present.

Applications should strive to be compatible across all Macintoshes, but there are times when an application must have knowledge about the machine that it is running on. The new trap, `SysEnvirons`, will give an application most of the information that it requires (what hardware, what version of system software...).

Using SysEnvirons

In most cases, if you examine why you want to test for the existence of a specific trap, you will find that there is an alternative method, for example:

I need to see if the "new" TextEdit calls are available.

Call `SysEnvirons` and check to see that `SysEnvRec.machineType >= 0` (128K ROMs or newer) and that we are running System 4.1 or later (System 4.1 and later support the new TextEdit on 128K and greater ROM machines—we can check this by just seeing if the `SysEnvirons` trap exists, if we get an `envNotPresent` error, we know it doesn't). In Pascal:

```
CONST
      CurrentVersion = 1;  {Current version of SysEnvirons}
VAR
      newTextAvail  : BOOLEAN;
      theWorld      : SysEnvRec;
BEGIN
{
This code checks to see if System 4.1 or later is running by calling
SysEnvirons. If SysEnvirons returns an envNotPresent error, we know that
we are running a system prior to 4.1, so we know we don't have the new
TextEdit. If SysEnvirons doesn't return envNotPresent, we check machine
type to make sure we aren't running on 64K ROMs (note: we assume that
envMachUnknown doesn't have 64K ROMs when we check machineType >= 0)
}
      IF SysEnvirons(CurrentVersion,theWorld) = envNotPresent THEN
            newTextAvail:= FALSE
```

```
        ELSE
                newTextAvail:= (theWorld.machineType >= 0);
END;
```

In C:

```
/* Current version of SysEnvirons */
#define       CurrentVersion      1
{
        Boolean         newTextAvail;
        SysEnvRec       theWorld;

/*
        see comment in the above Pascal
*/
if (SysEnvirons(CurrentVersion,&theWorld) == envNotPresent)
                newTextAvail = false;
        else
                newTextAvail = (theWorld.machineType >= 0);
}
```

I need to see if `PopUpMenuSelect` is implemented.

The same answer as above applies here, since the "new" Menu Manager calls are only implemented in System 4.1 on 128K or larger ROM machines (and, as we found above, `PopUpMenuSelect` has the same trap number as `Rename`, so calling `NGetTrapAddress` won't work on 64K ROMs).

If you find that you need information that is not contained in `SysEnvirons`, please send suggestions for extending it to Macintosh Developer Technical Support at the address in Technical Note #0.

Checking for Specific Functionality

There are rare times when you may feel that it is necessary to test for specific functionality. In order to allow for testing of specific trap functionality, there is an official unimplemented trap. This trap (`$A89F`) is unimplemented on all Macintoshes. To test to see if a particular trap that you wish to use is implemented, you can compare its address with the address of the unimplemented trap. Here are two fragments that show how to check to see if `Shutdown` is implemented. First, Pascal:

```
CONST
  ShutDownTrapNum    = $95;{trap number of Shutdown}
  UnImplTrapNum      = $9F;{trap number of "unimplemented trap"}

VAR
  ShutdownIsImplemented    : BOOLEAN;  {is Shutdown implemented}

BEGIN
{Is Shutdown implemented?}
        ShutdownIsImplemented :=
                NGetTrapAddress(ShutDownTrapNum,ToolTrap) <>
                NGetTrapAddress(UnImplTrapNum,ToolTrap);
END;
```

Here's a C fragment:

```
/*trap number of Shutdown*/
#define       ShutDownTrapNum     0x95
/*trap number of "unimplemented trap"*/
#define       UnImplTrapNum       0x9F

{
       Boolean ShutdownIsImplemented;

       /*Is Shutdown implemented?*/
       ShutdownIsImplemented =
              NGetTrapAddress(ShutDownTrapNum,ToolTrap) !=
              NGetTrapAddress(UnImplTrapNum,ToolTrap);
}
```

`NGetTrapAddress` is used because it ensures that you will get the correct trap in case there is a `ToolTrap` and an `OSTrap` with the same number. Please note that calling `NGetTrapAddress` does not cause compatibility problems with 64K ROMS. When run on those ROMs, it just becomes a `GetTrapAddress` call. You have to be careful on 64K ROMs—you can't test for `PopUpMenuSelect` (`$A80B`), for example, because it has the same trap number as `Rename`(`$A00B`). The 64K ROM didn't really differentiate between `ToolTraps` and `OSTraps` (there was **no** overlap in trap numbers). So, if you wanted to test for `PopUpMenuSelect`, you would need to first check to make sure you weren't running on 64K ROMs (see below).

You can get the trap number of the trap you wish to test for from *Inside Macintosh* (Appendix C of Volumes I-III and Appendix B of Volume IV). You can tell if the trap is an `OSTrap` or a `ToolTrap` by checking to see if bit 11 in the trap word is set, that is, traps like `$A8xx` (or `$A9xx` or `$AAxx`) that have the "8" component set, are `ToolTraps` and traps that don't (`$A0xx`) are `OSTraps`. The trap number that you pass to `NGetTrapAddress` for `ToolTraps` is the low 10 bits of the trap word (the trap number for `PopUpMenuSelect[$A80B]` is `$00B`). The trap number that you pass to `NGetTrapAddress` for `OSTraps` is the low 8 bits of the trap word (the trap number for `MoveHHi[$A064]` is `$064`).

`Shutdown` (`$A895`) is just an example of a trap that we might need to check before calling. Most applications won't call `ShutDown`, so this is just an example of how to do the testing.

#157: Problem with GetVInfo

See also: File Manager

Written by: Jim Friedlander
Updated:

September 1, 1987
March 1, 1988

The high-level call `GetVInfo` (and its low-level counterpart `PBGetVInfo`) may return inaccurate results for `freeBytes` when running HFS.

The high-level File Manager call `GetVInfo` returns the number of free bytes on a volume as one of its parameters. Since `GetVInfo` is really only glue that fills in a parameter block for you and then calls `PBGetVInfo`, the values returned from it are subject to the limitations (imposed for MFS) discussed in the File Manager chapter of *Inside Macintosh Volume IV* (p. 130): "**Warning**: `IOVNmAlBlks` and `ioVFrBlks`, which are actually unsigned integers, are clipped to 31744 (`$7C00`) regardless of the size of the volume." This will be fixed in future versions of the glue (newer than MPW 2.0.2), but for now, you need to call `PBHGetVInfo` yourself instead, as shown below.

The value that `GetVInfo` returns in `freeBytes` (`ioVFrBlks * ioVAlBlkSize`) will thus be **less than or equal to** the actual number of free bytes on a volume. This isn't catastrophic, but can be highly inconvenient if you really need to know how much free space is on a given volume.

Note: `IOVNmAlBlks` returned from `PB[H]GetVInfo` does not reflect the actual total number of blocks on an HFS disk, but rather only the blocks that are "available" for program use (it doesn't count blocks used for catalog information).

Here are two functions (one in MPW Pascal, one in MPW C) that return the actual number of free bytes on a volume, regardless of File System (if MFS is running, the `PBHGetVInfo` call will actually generate a `PBGetVInfo` call which will return the proper values for MFS volumes):

In MPW Pascal:

```
FUNCTION FreeSpaceOnVol(vRef:Integer; VAR freeBytes:Longint): OSErr;
TYPE
        {we need this to convert an unsigned integer to an unsigned
        longint}
        TwoIntsMakesALong    = RECORD
                                    CASE Integer OF
                                    1:      (long: LongInt);
                                    2:      (ints: ARRAY [0..1] OF Integer);
                                    END; {TwoIntsMakesALong}
```

```
VAR
        HPB             : HParamBlockRec;
        convert         : TwoIntsMakesALong;
        err             : OSErr;

BEGIN {FreeSpaceOnVol}
        WITH HPB DO BEGIN {set up parameter block for the PBGetVInfo call}
                ioNamePtr := NIL;   {we don't care about the name}
                ioVRefNum := vRef;  {this was passed in as a parameter}
                ioVolIndex := 0;    {use ioVRefNum only}
        END;                            {WITH}
        err := PBHGetVInfo(@HPB,false);
        FreeSpaceOnVol:= err;       {return error from HGetVInfo}

{
This next section needs some explanation.  ioVFrBlk is an unsigned
integer.  If we were to assign it(or coerce it) to a longint, it would get
sign extended by MPW Pascal and would thus be negative.  Since we don't
want that, we use a variant record that maps two integers into the space
of one longint.  The high word (convert.ints[0]) is cleared in the first
line, then the low word is assigned the value of HPB.ioVFrBlk.  The
resulting longint (convert.long) is now a correctly signed (positive) long
integer representing the number of free blocks.  NOTE:  this is only
necessary if the number of free blocks on a disk exceeds 32767 ($7FFF).
}
        IF err = 0 THEN BEGIN
                convert.ints[0] := 0;
                convert.ints[1] := HPB.ioVFrBlk;
                        freeBytes:= convert.long * HPB.ioVAlBlkSiz;
                END ELSE {PBHGetVInfo failed}
                        freeBytes:= 0; {return this if the routine failed}
END;  {FreeSpaceOnVol}
```

In MPW C:

```
OSErr freeSpaceOnVol(vRef,pfreeBytes)
short int vRef;
unsigned long int *pfreeBytes; /* C does this correctly!! */

{       /* freeSpaceOnVol */
        HVolumeParam HPB;
        OSErr err;

        HPB.ioNamePtr = 0L;         /* we don't care about the name */
        HPB.ioVRefNum = vRef;       /* this was passed in as a parameter */
        HPB.ioVolIndex = 0;         /* use ioVRefNum only */
        err = PBHGetVInfo(&HPB,false);
        if (err == noErr)
                *pfreeBytes = (unsigned long int)HPB.ioVFrBlk *
                                HPB.ioVAlBlkSiz;
        else
                *pfreeBytes = 0L;   /* return this if the routine failed */
        return(err);                /* function result */
}       /* freeSpaceOnVol */
```

#158: Frequently Asked MultiFinder Questions

See also: Technical Note #129—SysEnvirons
Technical Note #156—Checking For Specific Functionality
MultiFinder Developer's Package

Written by: Jim Friedlander September 1, 1987
Updated: March 1, 1988

This technical note provides answers to some of the more frequently asked questions about MultiFinder. The development name for MultiFinder was Juggler, so the term "juggle" is used in this technical note to denote a context switch.

How can I tell if WaitNextEvent is implemented?

Most applications should **not** need to tell if MultiFinder is running. Most of the time, the application really needs to know something like: "How can I tell if `WaitNextEvent` is implemented?" Here's a Pascal fragment that demonstrates how to check to see if `WaitNextEvent` is implemented:

```
FUNCTION TrapAvailable(tNumber: INTEGER; tType: TrapType): BOOLEAN;

   CONST
      UnimplementedTrapNumber = $A89F;  {number of "unimplemented trap"}

BEGIN {TrapAvailable}

{Check and see if the trap exists.}
{On 64K ROM machines, tType will be ignored.}

   TrapAvailable := ( NGetTrapAddress(tNumber, tType) <>
                      GetTrapAddress(UnimplementedTrapNumber) );

END;  {TrapAvailable}

FUNCTION WNEIsImplemented: BOOLEAN;

   CONST
      WNETrapNumber = $A860; {trap number of WaitNextEvent}

   VAR
      theWorld      : SysEnvRec; {to check if machine has new traps}
      discardError  : OSErr; {to ignore OSErr return from SysEnvirons}
```

```
BEGIN {WNEIsImplemented}

{  Since WaitNextEvent and HFSDispatch both have the same trap
   number ($60), we can only call TrapAvailable for WaitNextEvent
   if we are on a machine that supports separate OS and Toolbox
   trap tables. We call SysEnvirons and check if machineType < 0.}

   discardError := SysEnvirons(1, theWorld);

{  Even if we got an error from SysEnvirons, the SysEnvirons glue
   has set up machineType.}

   IF theWorld.machineType < 0 THEN
      WNEIsImplemented := FALSE
      {this ROM doesn't have separate trap tables or WaitNextEvent}
   ELSE
      WNEIsImplemented := TrapAvailable(WNETrapNumber, ToolTrap);
      {check for WaitNextEvent}

END;  {WNEIsImplemented}

{Note that we call SystemTask if WaitNextEvent isn't available.}

...
   hasWNE := WNEIsImplemented;
...
   IF hasWNE THEN BEGIN
      {call WaitNextEvent}
      ...
   END ELSE BEGIN
      {call SystemTask and GetNextEvent}
      ...
   END;
...
```

Here's a C fragment:

```
Boolean
TrapAvailable(tNumber, tType)
short    tNumber
TrapType tType
{

/* define trap number for old MPW or non-MPW C */
#ifndef _Unimplemented
#define _Unimplemented 0xA89F
#endif

/* Check and see if the trap exists. */
/* On 64K ROM machines, tType will be ignored. */

   return( NGetTrapAddress(tNumber, tType) !=
           GetTrapAddress(_Unimplemented) );

}
```

```
Boolean
WNEIsImplemented()
{

/* define trap number for old MPW or non-MPW C */
#ifndef _WaitNextEvent
#define _WaitNextEvent 0xA860
#endif

   SysEnvRec theWorld; /* used to check if machine has new traps */

/* Since WaitNextEvent and HFSDispatch both have the same trap
   number ($60), we can only call TrapAvailable for WaitNextEvent
   if we are on a machine that supports separate OS and Toolbox
   trap tables. We call SysEnvirons and check if machineType < 0. */

   SysEnvirons(1, &theWorld);

/* Even if we got an error from SysEnvirons, the SysEnvirons glue
   has set up machineType. */

   if (theWorld.machineType < 0) {
      return(false)
      /* this ROM doesn't have separate trap tables or WaitNextEvent */
   } else {
      return(TrapAvailable(_WaitNextEvent, ToolTrap));
      /* check for WaitNextEvent */
   }

}

/* Note that we call SystemTask if WaitNextEvent isn't available. */

...
   hasWNE = WNEIsImplemented();
...
   if (hasWNE) {
      /* call WaitNextEvent */
      ...
   } else {
      /* call SystemTask and GetNextEvent */
      ...
   }
...
```

Note: Testing to see if `WaitNextEvent` is implemented is **not** the same as testing to see whether MultiFinder is running. Systems 6.0 and newer include `WaitNextEvent` whether or not MultiFinder is running.

How can I tell if the MultiFinder Temporary Memory Allocation calls are implemented?

The technique that's used to determine this is similar to the above technique. The TrapAvailable routine above is reused. In Pascal:

```
FUNCTION TempMemCallsAvailable: BOOLEAN;

   CONST
      OSDispatchTrapNumber = $A88F; {number for temporary memory calls}

BEGIN {TempMemCallsAvailable}

{  Since OSDispatch has a trap number that was always defined
   to be a toolbox trap ($8F), we can always call TrapAvailable.
   If we are on a machine that does not have separate OS and
   Toolbox trap tables, we'll still get the right trap address.}

   TempMemCallsAvailable := TrapAvailable(OSDispatchTrapNumber, ToolTrap);
   {check for OSDispatch}

END;  {TempMemCallsAvailable}
```

In C:

```
Boolean
TempMemCallsAvailable()
{

/* define trap number for old MPW or non-MPW C */
#ifndef _OSDispatch
#define _OSDispatch 0xA88F
#endif

/* Since OSDispatch has a trap number that was always defined to
   be a toolbox trap ($8F), we can always call TrapAvailable.
   If we are on a machine that does not have separate OS and
   Toolbox trap tables, we'll still get the right trap address. */

   return(TrapAvailable(_OSDispatch, ToolTrap));
   /* check for OSDispatch */

}
```

How can I tell if my application is running in the background?

To run in the background under MultiFinder, an application must have set the `canBackground` bit (bit 12 of the first word) in the SIZE resource. In addition, the `acceptSuspendResumeEvents` bit (bit 14) should be set. An application can tell it is running in the background if it has received a suspend event but not a resume event.

When exactly does juggling take place?

Juggling takes place at `WaitNextEvent/GetNextEvent/EventAvail` time. If you have the `acceptSuspendResumeEvents` bit set in the SIZE resource, you will receive suspend/resume events. When you get a suspend event (or, when you call `EventAvail` and a suspend event has been posted), you will be juggled out the next time that you call `WNE/GNE/EventAvail`. When you receive a suspend event, you are going to be juggled, so don't do anything to try to retain control (such as calling `ModalDialog`).

Speaking of `ModalDialog`, MultiFinder will **not** suspend your application when the frontmost window is a modal dialog, though background tasks will continue to get time.

Can I disable suspend/resume events by passing the appropriate event mask to `WNE/GNE/EventAvail`?

suspend/resu,e events are not queued, so be careful when masking out `app4Evts`. You will still get the event, all that will happen if you mask out `app4Evts` is that your application won't know when it is going to be juggled out (your application will still be juggled out when you call `WNE/GNE/EventAvail`). If your application sets a boolean to tell whether or not it's in the foreground or the background, you definitely don't want to mask out `app4Evts`.

Should my application use WaitNextEvent?

Yes, this will enable background tasks to get as much time as possible. All user events that your program needs to handle will be passed to your application as quickly as possible. Applications that run in the background should try to be as friendly as possible. It's best to do things a small chunk at a time so as to give maximum time to the foreground application. "Cooperative multi-tasking" requires cooperation!

If your application calls `WaitNextEvent`, it shouldn't call `SystemTask`.

Is there anything else that I can do to be MultiFinder friendly?

It is very important that you save the positions of windows that you open, so that the next time the user launches your application, the windows will go where they had them last. This greatly enhances the usability of MultiFinder. With data files, the window positions can be stored in either the resource or the data fork.

If you have windows that aren't data windows (i.e. separate files), you can store information about their positions in one of two ways: in a separate configuration file or in a resource in your application. Using a separate configuration file is necessary if your application is shareable on AppleShare, since resource forks are not. The configuration file should be put in the folder that contains the currently open system folder (this is guaranteed to be a local, non-shared volume as opposed to a server volume). The `vRefNum/WDRefNum` of this folder can be obtained by calling `SysEnvirons` (`SysEnvRec.sysVRefNum`).

Can I use a debugger with MultiFinder?

Yes, MacsBug will load normally, since it is loaded well before MultiFinder. Since TMON is currently installed as a startup application, you should Set Startup to it, then launch MultiFinder manually (by holding down Option-Command while double-clicking the MultiFinder icon) or use a program that will run multiple startup applications (such as Sequencer), making sure that TMON is run before MultiFinder. If you try to run TMON after MultiFinder has been installed, a system crash will result. The latest version of TMON (2.8) has an INIT that loads it before MultiFinder is present.

It is necessary to check `CurApName` (`$910`) when you first enter a debugger (TMON users can anchor a window to `$910`) to see which layer (whose code, which low-memory globals and so on) is currently executing, especially if you entered the debugger by pressing the interrupt button.

What happened to animated icons under MultiFinder?

Finders 6.0 and newer no longer use the mask that you supply in an ICN# to "punch a hole" in the desktop. Instead, the Finder uses a default mask that consists of a solid black copy of the icon with no hole.

How can I ensure maximal compatibility with MultiFinder?

If you follow the guidelines presented in the MultiFinder Developer's Package you will stand a very good chance of being fully compatible with MultiFinder.

#159: Hard Disk Hacking

See also: Technical Note #96—SCSI Bugs
Technical Note #134—Hard Disk Medic
The Device Manager

Written by: Bo3b Johnson
Updated:

September 1, 1987
March 1, 1988

For those of a technical bent with some extra time, you can build your own hard disk system from a cheap SCSI drive and a driver that you write. This is not a project for those short on time, so beware.

We often get questions regarding the feasibility of connecting a 'generic' SCSI drive to the Macintosh, usually the Macintosh II. It is possible to use a standard drive, but it is important to be aware that there is a reason why a fully assembled drive costs more. When buying a hard disk you have two choices:

1) buy a fully assembled drive, formatting and driver software included
2) buy the pieces necessary to assemble your own: the drive itself, power supply if needed, cables, and development system to write a driver and formatter.

The second choice will often appear to be cheaper, since you don't have to pay for a fancy case with a fancy label. However, you are also missing the chance to pay for some fancy software that took some fancy amounts of time to write.

Do not underestimate the difficulty of building your own hard disk. SCSI drives are only partially standardized so a driver written for one drive will probably not work (at least not well) on another drive. All drives come with a formatting utility that also contains a driver for reading and writing sectors to the disk. For example, the Apple drives come with a program called HD SC Setup. Most third-party drives have a similar utility that is specific to their drive. The formatting operation varies widely depending on the drive, and the driver also may have to know about specific timing problems with a given drive. HD SC Setup only supports the drives which we produce.

If you decide that you want to hack together your own drive, you will need to write this formatter/driver program. It is non-trivial, and this is part of what you pay for when you buy an off-the-shelf drive. If you have the time, you may save some money. If you are writing your own formatter/driver program we can help you with problems you run into, but you must be familiar with SCSI terminology, the SCSI Manager, and be able to use an assembly level debugger like Macsbug or TMon. You may run into timing difficulties that require the use of a logic analyzer or SCSI analyzer to resolve.

This may sound like it is hard to write your own driver. It is. This may sound like we are trying to scare you off from writing your own driver. We are.

Macintosh
Technical Notes

Developer Technical Support

#160: Key Mapping

Revised by: Jim Luther, Peter Edberg, & Imran Sayeed
Written by: Cameron Birse

February 1991
September 1987

This Technical Note describes the Macintosh family key code mapping scheme when running System file 4.1 and later. This Note also provides a "safe" method for remapping keyboards.
Changes since October 1990: Added a section on how `'KMAP'` resources are matched to specific ADB keyboard types and a section on the original Macintosh and Macintosh Plus keyboards.

Introduction

System file 4.1 introduced a change in the keystroke mapping mechanism for the Macintosh family. Originally, a keystroke caused an interrupt, and the interrupt handler dispatched the keycode to a translation routine pointed to by a low-memory pointer (`Key1Trans` or `Key2Trans`). The System used to install this routine at boot time, and developers would generally replace it in part or entirely to remap the keyboard. When the keycode was mapped, it was returned to the interrupt handler, which then posted the event. The System file contained both the translation routine and the key map in `'INIT'` resources `ID = 0` and `ID = 1`.

In all System files since 4.1, the low-memory pointers are still there, and the Macintosh Plus still calls them; however, Macintosh systems equipped with ADB do not call these low-memory pointers. The System preserves them so applications that call them can still use them to translate keycodes, but since System file 4.1, they point to a routine that implements a different mechanism.

ADB Keyboards

With multiple Apple Desktop Bus (ADB) keyboards, a mechanism is needed to map the different raw keycodes to a standard virtual keycode that can be mapped to ASCII and special character sets. This mapping is done in an effort to reduce keyboard hardware dependence, and the raw mapping routine uses a table which is resident in the System `'KMAP'` resource. Basically, the raw keycode is used to index into the `'KMAP'` table; the value at the indexed location in the `'KMAP'` is what gets returned as the virtual keycode.

The `'KMAP'` resource ID that matches the keyboard type is used if that `'KMAP'` resource is present. If a `'KMAP'` resource ID that matches the keyboard type cannot be found, then the system attempts to use `'KMAP'` resource `ID = 0`. If `'KMAP'` resource `ID = 0` cannot be found, then raw keycode to virtual keycode mapping does not occur. On all Macintosh systems later than the Macintosh Plus, `'KMAP'` resource `ID = 0` is in ROM. The global variable `KbdType` (a byte) contains the type of the last keyboard used.

KbdType	Keyboard Type
$01	Apple Keyboard (standard) and Apple Desktop Bus Keyboard (IIGS)
$02	Apple Extended Keyboard and Apple Extended Keyboard II
$04	Apple Keyboard, ISO
$05	Apple Extended Keyboard II, ISO
$06	Portable
$07	Portable, ISO
$08	Apple Keyboard II
$09	Apple Keyboard II, ISO

With this mechanism, the keystroke causes an interrupt, and the interrupt handler maps the raw keycode to a "virtual" keycode , which is then sent to the `_KeyTrans` trap. This trap maps the virtual keycode to an ASCII value (using tables which reside in the `'KCHR'` resource of the System file) and returns that value to the handler which posts the event.

```
FUNCTION KeyTrans(transData:Ptr;keycode:INTEGER;VAR state: LONGINT):LONGINT
```

The `transData` parameter is a pointer to the `'KCHR'` image in memory. The `keycode` parameter is an integer composed of the modifier flags in bits 8-15, an up or down stroke flag in bit 7 (1=up), and the virtual key code in bits 6-0. The `state` parameter is a value internal to `_KeyTrans` which should be preserved across calls if dead keys are desired. It is dependent on the `'KCHR'` information, so if the `'KCHR'` is changed, state should be reset to zero.

The `LONGINT` returned is actually two 16-bit characters to be posted as events (usually the high byte of each is zero), high word first. A returned value of zero in either word should not be posted. Do not depend upon the word in which the character is returned; if both words are valid, then the high word should be posted first.

To remap the keyboard, one must supply a `'KCHR'` resource and have the System use it. Each `'KCHR'` resource has an associated `'SICN'` resource. The `'SICN'` resource provides a graphic representation of the current keyboard mapping. For example, the French keyboard layout has a `'SICN'` of a French flag to designate that particular map is currently active. The `'SICN'` resource should be some representation of the particular remap, and its ID number must be the same as that of the `'KCHR'`. The `'KCHR'` resource must be named appropriately, as it can be displayed in a scrolling list in the Keyboard Control Panel.

Macintosh and Macintosh Plus Keyboards

With the Macintosh (the original keyboard on the 128K and 512K Macintoshes) and Macintosh Plus keyboards, the event record contains the raw keycode, since there is no `'KMAP'` mapping. For the domestic Macintosh keyboard and the Macintosh Plus keyboard, this is not a problem, since the raw keycodes generated by those keyboards are identical to the virtual keycodes. This is not the case for the Macintosh international keyboard, which is still used with the Macintosh Plus on many international systems. For this keyboard, the event record contains a raw keycode which can not be treated as a virtual keycode.

If you need to obtain virtual keycodes for the Macintosh international keyboard, you need to map the raw keycode in the event record to a virtual keycode. The following table provides the necessary mapping (the raw keycodes generated by this keyboard are in the range $00-$3F; keycodes above this are generated by the optional keypad that may be used with this keyboard). If the raw keycode is used as an offset into this table, the byte at that offset is the virtual keycode.

This mapping is also performed by the _Key1Trans hook before it calls _KeyTrans, if the keyboard is a Macintosh international type.

```
oldIntlKeybdRawToVirtual                                    ; raw keycode:
        dc.b  $00, $01, $02, $03, $04, $05, $32, $06   ; $00 .. $07
        dc.b  $07, $08, $2c, $09, $0c, $0d, $0e, $0f   ; $08 .. $0f
        dc.b  $10, $11, $12, $13, $14, $15, $16, $17   ; $10 .. $17
        dc.b  $18, $19, $1a, $1b, $1c, $1d, $1e, $1f   ; $18 .. $1f
        dc.b  $20, $21, $22, $23, $2a, $25, $26, $27   ; $20 .. $27
        dc.b  $28, $29, $24, $2e, $2f, $0b, $2d, $2b   ; $28 .. $2f
        dc.b  $30, $34, $0a, $33, $31, $35, $36, $37   ; $30 .. $37
        dc.b  $38, $39, $3a, $3b, $3c, $3d, $3e, $3f   ; $38 .. $3f
```

The global variable KbdType (a byte) contains the type of the last keyboard used. The following table shows the values of the global variable KbdType for the Macintosh and Macintosh Plus keyboards:

KbdType	Keyboard Type
$03	Macintosh (domestic or international)
$0B	Macintosh Plus

For both the Macintosh domestic and international keyboards, the global variable KbdType is 3. The Macintosh has no way to distinguish between these two keyboards, and must rely on the user to indicate which keyboard is being used by clicking on the appropriate picture in a panel in the Keyboard Cdev (this panel only appears on non-ADB Macintoshes). A byte flag in the 'itlc' resource ID = 0 indicates which keyboard the user has specified. If the KbdType global contains 3, you can test this flag to determine if the international version of the keyboard is being used (following is the assembly-language version):

```
with        ItlcRecord
subq        #4,sp                   ; space for returned handle
move.l      #'itlc',-(sp)           ; push itlc type
clr.w       -(sp)                   ; want ID=0
_GetResource                        ; get the itlc resource
move.l      (sp)+,d0                ; did we get it?
beq         myErrorHandling         ; if not, bail
move.l      d0,a0                   ; copy handle
move.l      (a0),a0                 ; get pointer
tst.b       itlcOldKybd(a0)         ; check flag
endwith   ;ItlcRecord
; if non-zero, international keyboard is being used
```

Hardware Dependencies

Although the principle underlying virtual keycodes is to have a standard keycode for a character regardless of the actual keyboard used, some hardware dependent differences are still present, and covered in this section.

- The virtual keycodes for the cursor keys and for some keypad operator keys are different on the ADB keyboards and the non-ADB keyboards:

Key Description	ADB Keycode	Non-ADB Keycode
left arrow	$7B	$46
right arrow	$7C	$42
down arrow	$7D	$48
up arrow	$7E	$4D
keypad plus sign (+)	$45	$46 (with Shift bit set in modifiers)
keypad asterisk (*)	$43	$42 (with Shift bit set in modifiers)
keypad equal sign (=)	$51	$48 (with Shift bit set in modifiers)
keypad slash (/)	$4B	$4D (with Shift bit set in modifiers)

  Notice that on non-ADB keyboards, the keycodes for the keypad operators listed duplicate the keycodes for the cursor keys. On these keyboards, holding Shift and pressing Left Arrow produce the plus sign character (+), for example.

- The Macintosh International keyboard and the ISO ADB keyboards have an extra key that is not present on the domestic keyboards. This key produces virtual keycode $0A.

- It is possible to reassign the virtual keycodes for the Shift, Option, and Control keys on the right side of the ADB Extended keyboards. Please refer to *Inside Macintosh*, V-193, The Toolbox Event Manager, for an explanation.

- There is a different virtual keycode for the Enter key depending on whether it is on the keypad (as on the Macintosh Plus keyboard and most ADB keyboards) or on the main section of the keyboard (as on the Macintosh keyboard and the Portable keyboard). The keypad version has keycode $4C, while the main keyboard version has keycode $34.

Remapping the Keyboard

Remapping the keyboard can be done two ways: either at boot time or from within an application. Remapping from within an application can be made permanent (until the next boot) or only for the life of the application. The remapping is accomplished by modifying a `'KCHR'` resource and telling the System to use the new `'KCHR'`. The `'KCHR'` must have an ID number in the range of the appropriate script and must have an associated `'SICN'` resource with the same ID number. The Roman script, for example, uses the range 0 to 16383, and the standard `'KCHR'` IDs for each country are the same as the country code (e.g., US = 0, French = 1, German = 2, etc.). Other Roman keyboards should have numbers somewhere in the script range (e.g., "Dvorak" at ID 500).

Remap At Boot Time

To remap the keyboard at boot time, there must be a modified `'KCHR'` resource in the System file with an ID number in the range of the appropriate script and an associated `'SICN'` resource with

the same ID number. To make the System use the modified 'KCHR' at boot time, one must change the entries in the 'itlb' resource in the System file to reflect the ID of the modified 'KCHR' and 'SICN' resources. Refer to the latest version of MPW's SysTypes.r file for the exact format of the 'itlb' resource.

Since Apple's System Software Licensing policy forbids shipping a modified System file, Apple suggests using the Installer to install the new resources. This installation should assign the new resources their IDs based upon what is currently in the System file to avoid all conflicts. Refer to Technical Note #75, Apple's Multidisk Installer for more details on Apple's Installer program.

Remap After Boot Time

To remap the keyboard after boot time, there must be a modified 'KCHR' resource in the System file or in the application with an ID number in the range of the appropriate script and an associated 'SICN' resource with the same ID number. One must also call the Script Manager to set up the proper resources and tell the System to use them. First call _SetScript to set the Script Manager's global variable for the 'KCHR' resource ID, then call _SetScript again to set the global variable for the 'SICN' resource ID. Now call _KeyScript to load the resources and set up the System to use them. The following sample code demonstrates these calls:

MPW Pascal

```
CONST
  DvorakID = 500;

VAR
  err: OSErr;

BEGIN
  err := SetScript(smRoman, smScriptKeys, DvorakID);
  err := SetScript(smRoman, smScriptIcon, DvorakID);
   KeyScript(smRoman);
END;
```

'KCHR' Resource Format

This section provides a general description of the 'KCHR' resource format; refer to the latest version of MPW's SysTypes.r file for the exact format of the 'KCHR' resource.

The 'KCHR' resource consists of a two-byte version number followed by a 256-byte modifier table, mapping all 256 possible modifier states to a table number. This table is followed by a two-byte count of tables, which is, in turn, followed by that many 128-byte ASCII tables. The ASCII tables map the virtual keycode to an ASCII value; zero signifies a dead key, and in this case the dead key table must be searched. The dead key table is composed of a count of dead key records (two bytes) and that many dead key records. A dead key record consists of a one-byte table number, a one-byte virtual keycode (without an up or down bit), a completor table, and a no-match character.

When _KeyTrans searches the dead key records, it checks for a match with the table number and the keycode. If there is no match, it is not a dead key, and a zero is returned. If there is a match, it is recorded in the state variable. If the previous key was a dead key, the completor table is searched. The completor table is comprised of a count of completor records, followed by that number of completor records.

A completor record is simply a substitution list for ASCII characters. If the ASCII character matches the first byte in the completor record, the second byte is substituted for it. When there is no substitution to be made, the original ASCII character is preceded by the no match character found at the at the end of the dead key record.

'KMAP' Resource Format

This section provides a general description of the `'KMAP'` resource format; refer to the latest version of MPW's SysTypes.r file for the exact format of the `'KMAP'` resource.

The 'KMAP' resource starts with a two-byte ID, followed by a two-byte version number. These four bytes are followed by the 128-byte keycode mapping table, described previously. The table is followed by a list of exceptions. The 128-byte table is simply a one-to-one mapping of real keycodes to virtual keycodes; the first byte is the virtual keycode for $00, the second for $01, etc. The high bit of the virtual keycode signals an exception entry in the exception list.

The exception list is used to enable the device driver to initiate communication with the device, usually to perform a state change. The exception list begins with a two-byte record count followed by that many records. The format of the exception record is available in the MPW SysTypes.r file. The raw keycode is the keycode as generated by the device. The `XOR` bit informs the driver to invert the state of the key instead of using the state provided by the hardware. This can be used to provide keys that lock in software. *Inside Macintosh*, Volume V, The Apple Desktop Bus, describes the ADB opcode. Finally, the data string is a Pascal string that is passed to the `_ADBOp` trap.

Further Reference:

- *Inside Macintosh*, Volume V, The Toolbox Event Manager
- *Inside Macintosh*, Volume V, The Apple Desktop Bus
- Technical Note #75, Apple's Multidisk Installer
- Technical Note #263, International Canceling

Macintosh
Technical Notes



Developer Technical Support

#161: A Printing Loop That Cares...

Revised by: Pete "Luke" Alexander October 1990
Written by: Ginger Jernigan September 1987

This Technical Note discusses opening and closing the Printing Manager with calls to `_PrOpen` and `_PrClose` as well as how to handle errors at print time.
Changes since October 1989: Added the section on error checking, incorporating the error code descriptions formerly found in Technical Note #72, Optimizing For The LaserWriter—Techniques and an updated version of the information formerly found in Technical Note #118, How To Check and Handle Printing Errors.

Introduction

At one time, Apple recommended that developers call `_PrOpen` at the beginning of their application and `_PrClose` at the end, before returning to the Finder. This recommendation was in the ancient past when an application only had to deal with a single printer driver.

As more printer drivers became available, it became important for an application to consider the presence of other applications and how opening and closing the printer driver affected them. The user could open the Chooser at any time and change the current printer driver without the current application's knowledge. If an application followed the old philosophy and a user changed the current printer driver while running the application, the next time the user attempted to print, the wrong driver would be open, the Printing Manager would not be able to find the necessary resources, and the user would get an error.

The Current Recommendation

DTS currently recommends that applications open and close the printer driver **each** time the application uses the Printing Manager.

MPW Pascal

```
{------ PrintStuff ------------------------------------------------------------}
{**
 **    PrintStuff will call all of the necessary Print Manager calls to print
 **    a document.  It checks PrError() after each Print Manager call.  If an
 **    error is found, all of the Print Manager open calls (i.e., PrOpen,
 **    PrOpenDoc...) will have a corresponding close call before the error
 **    is posted to the user.  You want to use this approach to make sure the
 **    Print Manager closes properly and all temporary memory is released.
 **}
```

```
PROCEDURE PrintStuff;

VAR
  copies,
  firstPage,
  lastPage,
  loop,
  numberOfCopies,
  pageNumber,
  printmgrsResFile,
  realNumberOfPagesInDoc  : Integer;
  PrintError              : LongInt;
  oldPort                 : GrafPtr;
  thePrRecHdl             : THPrint;
  thePrPort               : TPPrPort;
  theStatus               : TPrStatus;

BEGIN
  GetPort(oldPort);

  {**
      UnLoadTheWorld will swap out ALL unneeded code segments and data that are NOT required
      during print time. Your print code must be a separate code segment.
  **}
  UnLoadTheWorld;

  thePrRecHdl := THPrint(NewHandle(SIZEOF(TPrint)));

  IF (MemError = noErr) AND (thePrRecHdl <> NIL) THEN
    BEGIN
      PrOpen;
        IF (PrError = noErr) THEN
          BEGIN
            {**
                Save the current resource file (i.e. the printer driver's)
                so the driver will not lose its resources upon return from
                the pIdleProc.
             **}
            printmgrsResFile := CurResFile;
            PrintDefault(thePrRecHdl);

            IF (PrError = noErr) THEN
              BEGIN
                IF (PrStlDialog(thePrRecHdl)) THEN
                  BEGIN
                    {**
                        DetermineNumberOfPagesinDoc determines the number of
                        pages contained in the document by comparing the size of
                        the document with rPage from the TPrInfo record (IM II-150).
                        It returns the number of pages required to print the
                        document for the currently selected printer.
                     **}

                    realNumberOfPagesinDoc := DetermineNumberOfPagesinDoc
                                                (thePrRecHdl^^.prInfo.rPage);

                    IF (PrJobDialog(thePrRecHdl)) THEN
                      BEGIN
                        {**
                            Get the number of copies of the document that the
                            user wants printed from iCopies of the TPrJob record
                            (IM II-151).
                         **}

                        numberOfCopies := thePrRecHdl^^.prJob.iCopies;
```

```
{**
    Get the first and last pages of the document that
    were requested to be printed by the user from iFstPage
    and iLastPage from the TPrJob record (IM II-151).
 **}

firstPage := thePrRecHdl^^.prJob.iFstPage;
lastPage := thePrRecHdl^^.prJob.iLstPage;

{**
    Print "all" pages in the print loop
 **}

thePrRecHdl^^.prJob.iFstPage := 1;
thePrRecHdl^^.prJob.iLstPage := 9999;

{**
    Determine the "real" number of pages contained in
    the document.  Without this test, you would print
    9999 pages.
 **}

IF (realNumberOfPagesinDoc < lastPage) THEN
  lastPage := realNumberOfPagesinDoc;

PrintingStatusDialog := GetNewDialog(257, NIL, POINTER(-1));

{**
    Print the number of copies of the document requested by the user
    from the Print Job Dialog.
 **}

For copies := 1 To numberOfCopies Do
  BEGIN

    {**
        Install a pointer to your pIdle proc in my print record.
     **}
     thePrRecHdl^^.prJob.pIdleProc := @checkMyPrintDialogButton;

    {**
        Restore the resource file to the printer driver's.
     **}
    UseResFile(printmgrsResFile);

    thePrPort := PrOpenDoc(thePrRecHdl, NIL, NIL);

    IF (PrError = noErr) THEN
      BEGIN
        {**
            Print the range of pages of the document requested
            by the user from the Print Job Dialog.
         **}
        pageNumber := firstPage;
        WHILE ((pageNumber <= lastPage) AND (PrError = noErr)) DO
          BEGIN

            PrOpenPage(thePrPort, NIL);

              IF (PrError = noErr) THEN
                BEGIN
```

```
                                        {**
                                            rPage (IM II-150) is the printable
                                            area for the currently selected printer.
                                            By passing the current enables your app
                                            to use the same routine to draw to the
                                            screen and the printer's GrafPort.
                                         **}

                                        DrawStuff (thePrRecHdl^^.prInfo.rPage,
                                                   GrafPtr (thePrPort),
                                                   pageNumber);
                                    END;
                                PrClosePage(thePrPort);
                                pageNumber := pageNumber + 1;
                            END;  {**  End pagenumber loop  **}
                        END;
                      PrCloseDoc(thePrPort);
                    END;  {**  End copies loop  **}

                    {**
                        The printing job is being canceled by the request of
                        the user from the Print Style Dialog or the Print Job
                        Dialog.  PrError will be set to iPrAbort to tell the
                        Print Manager to abort the current printing job.
                     **}
                END
                  ELSE
                PrSetError(iPrAbort);   {**  Cancel from the job dialog  **}
            END
              ELSE
            PrSetError(iPrAbort);   {**  Cancel from the style dialog  **}
        END;
      END;
  IF (thePrRecHdl^^.prJob.bJDocLoop = bSpoolLoop) and (PrError = noErr) THEN
    PrPicFile(thePrRecHdl, NIL, NIL, NIL, theStatus);

  {**
      Grab the printing error before you close the Print Manager and the error disappears.
   **}

  PrintError := PrError;

  PrClose;

  {**
      You do not want to report any printing errors until you have fallen
      through the printing loop. This will make sure that ALL of the Print
      Manager's open calls have their corresponding close calls, thereby
      enabling the Print Manager to close properly and that all temporary
      memory allocations are released.
   **}

  IF (PrintError <> noErr) THEN
    PostPrintingErrors (PrintError);

  END;

IF (thePrRecHdl <> NIL) THEN
  DisposHandle(Handle (thePrRecHdl));

IF (PrintingStatusDialog <> NIL) THEN
  DisposDialog(PrintingStatusDialog);

SetPort(oldPort);
END;  {**  PrintStuff  **}
```

MPW C

```
/*------ PrintStuff ---------------------------------------------------------*/
 **
 **    PrintStuff will call all of the necessary Print Manager calls to print
 **    a document. It checks PrError() after each Print Manager call. If an error
 **    is found, all of the Print Manager open calls (i.e., PrOpen, PrOpenDoc...)
 **    will have a corresponding close call before the error is posted to the user.
 **    You want to use this approach to make sure the Print Manager closes properly
 **    and all temporary memory is released.
 **/

void PrintStuff ()
{
  GrafPtr      oldPort;
  short        copies,
               firstPage,
               lastPage,
               numberOfCopies,
               printmgrsResFile,
               realNumberOfPagesinDoc,
               pageNumber,
               PrintError;
  THPrint      thePrRecHdl;
  TPPrPort     thePrPort;
  TPrStatus    theStatus;

  GetPort(&oldPort);

 /**
     UnLoadTheWorld will swap out ALL unneeded code segments and data that
     are NOT required during print time. Your print code must be a separate
     code segment.
  **/

  UnLoadTheWorld ();
  thePrRecHdl = (THPrint)  NewHandle (sizeof (TPrint));

 /**
     Check to make sure that the memory manager did not produce an error
     when it allocated the print record handle and make sure it did not pass
     back a nil handle.
  **/

  if (MemError() == noErr && thePrRecHdl != nil)
    {
      PrOpen();

      if (PrError() == noErr)
        {
         /**
              Save the current resource file (i.e. the printer driver's) so
              the driver will not lose its resources upon return from the pIdleProc.
          **/
          printmgrsResFile = CurResFile();
          PrintDefault(thePrRecHdl);
```

```
if (PrError() == noErr)
  {
    if (PrStlDialog(thePrRecHdl))
      {
       /**
           DetermineNumberOfPagesinDoc determines the number of pages
           contained in the document by comparing the size of the document
           with rPage from the TPrInfo record (IM II-150). It returns the
           number of pages required to print the document for the currently
           selected printer.
        **/

        realNumberOfPagesinDoc = DetermineNumberOfPagesinDoc
                                    ((**thePrRecHdl).prInfo.rPage);

        if (PrJobDialog(thePrRecHdl))
          {
           /**
                Get the number of copies of the document that the user
                wants printed from iCopies of the TPrJob record (IM II-151).
            **/

            numberOfCopies = (**thePrRecHdl).prJob.iCopies;

             /**
                 Get the first and last pages of the document that
                 were requested to be printed by the user from iFstPage
                 and iLastPage from the TPrJob record (IM II-151).
              **/

            firstPage = (**thePrRecHdl).prJob.iFstPage;
            lastPage = (**thePrRecHdl).prJob.iLstPage;

             /**
                 Print "all" pages in the print loop
              **/

            (**thePrRecHdl).prJob.iFstPage = 1;
            (**thePrRecHdl).prJob.iLstPage = 9999;

             /**
                 Determine the "real" number of pages contained in the document.
                 Without this test, you would print 9999 pages.
              **/

            if (realNumberOfPagesinDoc < lastPage)
              lastPage = realNumberOfPagesinDoc;

            PrintingStatusDialog = GetNewDialog(257, nil, (WindowPtr) -1);

             /**
                 Print the number of copies of the document
                 requested by the user from the Print Job Dialog.
              **/
            for (copies = 1; copies <= numberOfCopies; copies++)
              {
               /**
                   Install a pointer to your pIdle proc in my print record.
                **/
               (**thePrRecHdl).prJob.pIdleProc = checkMyPrintDialogButton;

               /**
                   Restore the resource file to the printer driver's.
                **/
```

```
                        UseResFile(printmgrsResFile);
                        thePrPort = PrOpenDoc(thePrRecHdl, nil, nil);

                        if (PrError() == noErr)
                          {

                           /**
                               Print the range of pages of the document
                               requested by the user from the Print Job Dialog.
                            **/
                            pageNumber = firstPage;
                            while (pageNumber <= lastPage && PrError() == noErr)
                              {
                                PrOpenPage(thePrPort, nil);

                                if (PrError() == noErr)
                                  {
                                   /**
                                       rPage (IM II-150) is the printable area
                                       for the currently selected printer. By passing
                                       the current port to the draw routine, enables
                                       your app to use the same routine to draw to
                                       the screen and the printer's GrafPort.
                                    **/
                                    DrawStuff ((**thePrRecHdl).prInfo.rPage,
                                                (GrafPtr) thePrPort, pageNumber);
                                  }

                                PrClosePage(thePrPort);
                                pageNumber++;
                              } /**  End pageNumber loop  **/
                          }
                        PrCloseDoc(thePrPort);
                      } /**  End copies loop  **/
                  }
                 /**
                     The printing job is being canceled by the request of the
                     user from the Print Style Dialog or the Print Job Dialog.
                     PrError will be set to PrAbort to tell the Print Manager to
                     abort the current printing job.
                  **/
                else
                  PrSetError (iPrAbort);   /**  cancel from the job dialog  **/
              }
            else
              PrSetError (iPrAbort);   /**  cancel from the style dialog  **/
          }
      }

    if (((**thePrRecHdl).prJob.bJDocLoop == bSpoolLoop) && (PrError() == noErr))
      PrPicFile(thePrRecHdl, nil, nil, nil, &theStatus);

     /**
         Grab the printing error before you close the Print Manager
         and the error disappears.
      **/

    PrintError = PrError();

    PrClose();
```

```
        /**
            You do not want to report any printing errors until you have fallen
            through the printing loop. This will make sure that ALL of the Print
            Manager's open calls have their corresponding close calls, thereby
            enabling the Print Manager to close properly and that all temporary
            memory allocations are released.
         **/
        if (PrintError != noErr)
          PostPrintingErrors (PrintError);
      }

    if (thePrRecHdl != nil)
      DisposHandle((Handle) thePrRecHdl);

    if (PrintingStatusDialog != nil)
      DisposDialog(PrintingStatusDialog);

    SetPort(oldPort);
  }  /** PrintStuff **/
```

Checking And Handling Printing Errors

An application should always check for error conditions while printing by calling `PrError`. `PrError` returns errors from the Printing Manager (and some AppleTalk and OS errors) that occur during printing.

As the example code demonstrates, an application should call `PrError` after each call to a Printing Manager function or procedure. By consistently checking `PrError` after each call, the application is able to catch any errors created at print time and report them to a user via a dialog box in a clean and graceful manner.

The following section outlines some general error-handling guidelines.

- You should avoid calling `PrError` within your pIdle procedure; errors that occur while it is executing are usually temporary and serve only as internal flags for communication within the printer driver—they are not intended for the application. If you absolutely must call `PrError` within your idle procedure, and an error occurs, never abort printing within the idle procedure itself. Wait until the last called printing procedure returns, then check to see if the error still remains. Attempting to abort printing within an idle procedure is a guarantee of certain death.

- Upon detecting an error after the completion of a printing routine, stop drawing at that point, and proceed to the next procedure to close any previously made open calls. For example, if you detect an error after calling `PrOpenDoc`, skip to the next `PrCloseDoc`. Or, if you get an error after calling `PrOpenPage`, skip to the next `PrClosePage` and `PrCloseDoc`. Remember that if you have called `PrOpen`, then you must call the corresponding `PrClose` to ensure that printing closes properly and all temporary memory allocations are released and returned to the heap.

- Do not display any alert or dialog boxes to report an error until the end of the printing loop. Once at the end, check for the error again; if there is no error assume that printing completed normally. If the error is still present, then you can alert the user.

This technique is important for two reasons. First, if you display a dialog box in the middle of the printing loop, it could cause errors that can terminate an otherwise normal job. For example, if the printer is an AppleTalk printer, the connection can be terminated abnormally since the driver would be unable to respond to AppleTalk requests received from the printer while the dialog box was waiting for input from the user. If the printer does not hear from the Macintosh with a short period of time (e.g., 30 seconds), it times out, assuming that the Macintosh is no longer there, which results in a prematurely broken connection causing another error to which the application must respond.

In addition, the driver may have already displayed its own dialog box in response to an error. In this instance, the driver posts an error to let the application know that something went wrong and it should abort printing. For example, when the LaserWriter driver detects that the Laser Prep version which has been downloaded to the LaserWriter is different than that with which the user is trying to print, it displays the appropriate dialog box informing the user of the situation and giving him the option of reinitializing the printer. If the user chooses to cancel printing, the driver posts an error to let the application know that it needs to abort, but since the driver has already taken care of the error by displaying a dialog box, the error is reset to zero before the printing loop is complete. The application should check for the error again at the end of the printing loop, and if it still indicates an error, the application can then display the appropriate dialog box.

- If using `PrGeneral`, be prepared to receive the following errors: `NoSuchRsl`, `OpNotImpl`, and `resNotFound`. In all three cases, the application should be prepared to continue to print without using the features of that particular opcode.

  However, in the case of the `resNotFound` error, it means the current printer driver does not support `PrGeneral`. This lack of support should not be a problem for an application, but it needs to be prepared to deal with this error. If you receive a `resNotFound` error from `PrError`, clear the error with a call to `PrSetError(0)`; otherwise, `PrError` might still contain this error the next time you check it, which would prevent your application from printing.

Canceling or Pausing the Printing Process

If you install a procedure for handling requests to cancel printing, with an option to pause the printing process, beware of timeout problems when printing to the LaserWriter. Communication between the Macintosh and the LaserWriter must be maintained to prevent a job or a wait timeout. If there is no communication for a period of time (over two minutes), the printer times out and the print job terminates due to a wait timeout. Or, if the print job requires more than three minutes to print, the print job terminates due to a job timeout. Since, there is no good method to determine to what type of printer an application is printing, it is probably a good idea to document the possibility of a LaserWriter timing out for a user who chooses to select "pause" for over two minutes.

Error Messages Created In Print Land...

The Printing Manager reports the error messages covered in this section. If an error that does not belong to the Printing Manager occurs, the Printing Manager puts it into low memory, where it can be retrieved with a call to `PrError`, and terminates the printing loop, if necessary. As already documented, if you encounter an error in the middle of a printing loop, do not jump out; fall through the loop and let the Printing Manager terminate properly.

Error Code	Constant	Description
0	noErr	No error
128	iPrAbort	Abort the printing process (Command-period)
-1	iPrSavePFil	Problem saving print file
-17	controlErr	Unimplemented Control call
-27	iIOAbort	I/O problems
-108	iMemFullErr	Not enough heap space

The following errors are specific to the LaserWriter family:

-4101	Printer not found or closed
-4100	Connection just closed
-4099	Write request too big
-4098	Request already active
-4097	Bad connection refnum
-4096	No free Connect Control Blocks (CCBs) available
-8133	PostScript error occurred during transmission of data to printer. Most often caused by a bug in the PostScript code being downloaded.
-8132	Timeout occurred. This error is returned when no data has been sent to the printer for two minutes. Usually caused by extremely long imaging time.
-8131	Printer not responding: it may have been turned "off." This error occurs if a user turns off the LaserWriter in the middle of a print job.

The following errors are specific to `PrGeneral`:

1	NoSuchRsl	Requested resolution is not supported
2	OpNotImpl	Requested `PrGeneral` opcode not implemented in the current printer driver.
-192	resNotfound	The current printer driver does not support `PrGeneral`.

The most common error encountered is -4101, which is generated if no LaserWriter is selected. Since this error is so common, it is a good idea to display a dialog box requesting the user to select a printer from the Chooser when this error is encountered.

Further Reference:

- *Inside Macintosh*, Volume II-145 & V-410, The Printing Manager
- Technical Note #122, Device-Independent Printing
- *develop*, July 1990, Issue 3, "Meet PrGeneral"

#162: MPW 2.0 Pascal Compiler Bug

Written by: Jim Friedlander
Updated:

September 1, 1987
March 1, 1988

This note formerly described a bug in the MPW 2.0 Pascal compiler. This bug has been fixed in MPW 2.0.2.

Macintosh Technical Notes

#163: Adding Color With CopyBits

See also: Color QuickDraw

Written by: Chris Derossi
Updated:

November 2, 1987
March 1, 1988

Inside Macintosh Volume V states that the foreground and background colors are applied to an image during a `CopyBits` or `CopyMask` call. Accidental use of this feature can create bizarre coloring effects. This note explains what happens, how to avoid problems, and how to use it.

What Happens

Color QuickDraw has a feature that will allow you to convert a monochrome image to a color image. During a `CopyBits` or `CopyMask` call, if the foreground and background colors are not black and white, respectively, Color QuickDraw performs the following operation on every pixel being copied:

```
NOTE: color table index = pixel value

s = color table index of source pixel
fg = color table index of foreground color
bg = color table index of background color

ColoredPixelValue = (NOT(s) AND bg) OR (s AND fg)
```

If your source image contains only black and white pixels, then all black pixels would become the foreground color and all white pixels would become the background color. This is because the color table index for white is all zeros and the color table index for black is all ones.

For example, suppose your source image was a 4-bit deep color `PixMap`. Then the color table index for white (in binary) is 0000 and the index for black is 1111. And let's suppose that your foreground color is green with an index of 1101 while your background color is red with an index of 0011. Then for the black pixels, the above procedure produces:

```
ColoredPixelValue = (NOT(1111) AND 0011) OR (1111 AND 1101)
     1101         = (  0000    AND 0011) OR (1111 AND 1101)
```

And the operation on the white pixels yields:

```
ColoredPixelValue = (NOT(0000) AND 0011) OR (0000 AND 1101)
     0011         = (  1111    AND 0011) OR (0000 AND 1101)
```

Possible Problems

This colorizing will only work on 2-color (i.e. black and white) images, and then only if those colors occupy the first and last entries in the color table. Trying to colorize colors that are not the first and last color table entries will yield unexpected results.

This is mainly due to the fact that the colorizing algorithm uses a pixel's color table index value rather than its actual RGB color. To illustrate this, let's assume that foreground and background colors are as above, and your image contains yellow with a color table index of 1000. The colorizing operation would give:

```
ColoredPixelValue = (NOT(1000) AND 0011) OR (1000 AND 1101)
     1011         = (  0111    AND 0011) OR (1000 AND 1101)
```

Since the color table may have any RGB color at the resulting index position, the final color may not even be close to the source, foreground, or background colors.

Similar things occur if you are trying to colorize a black and white image when white and black do not occupy the first and last positions in the color table.

The bottom line rules for `CopyBits`ing in a color environment are these:

- Thou shalt set thy background color to white and thy foreground color to black before calling `CopyBits` or `CopyMask`, unless thou art coloring a monochrome image.
- Thou shalt, when colorizing, make sure that the first color table entry is white and the last color table entry is black.

The second rule is easy to follow because the default color tables are constructed properly, and if you are using the Palette Manager (and you are, right?) then it will make sure that the color tables obey this rule.

How To Colorize—An Example

This code fragment shows how to implement a color fill, like the paint bucket in MacPaint. It relies on three main things: `SeedCFill` for calculating the fill area, `CopyMask` for actually changing the bits, and QuickDraw colorizing.

```
PROCEDURE PaintBucket(where: Point; paintColor: RGBColor);

  VAR
       savedFG  : RGBColor;
       offBits  : BitMap;

  BEGIN
       {First, create an offscreen bitmap.}
       offBits.bounds := myWindow^.portRect;
       WITH offBits.bounds DO BEGIN
            offBits.rowBytes := ((right - left + 15) DIV 16) * 2;
            offBits.baseAddr := NewPtr((bottom-top) * offBits.rowBytes);
       END;

       {Check MemError here! Make sure NewPtr succeeded!}

       SeedCFill(myWindow^.portBits,offBits,myWindow^.portRect,
            myWindow^.portRect,where.h,where.v,NIL,0);
       GetForeColor(savedFG);
       RGBForeColor(paintColor);
       CopyMask(offBits,offBits,myWindow^.portBits,myWindow^.portRect,
            myWindow^.portRect,myWindow^.portRect);
       RGBForeColor(savedFG);

       DisposPtr(offBits.BaseAddr);
  END;
```

The variable `offBits` is an offscreen `BitMap` (not a `PixMap`) with `bounds = myWindow^.portRect`. `SeedCFill` effectively creates, in the offscreen `BitMap`, a monochrome image of the bits that we want to paint. Since `offBits` contains the exact bits that we want to paint, it is used as both the source image and the mask for `CopyMask`.

By setting the foreground color to the desired paint color, the result is a colorized version of the mask (the paint area) being copied onto the window's `PixMap` without affecting any other bits.

#164: MPW C Functions: To declare or not to declare, that is the question.

See also: MPW C Manual
Using Assembly Language
The C Programming Language, Kernighan & Ritchie
Technical Note 166—MPW C Functions
Using Strings or Points as Arguments

Written by: Fred A. Huxham November 2, 1987
Updated: March 1, 1988

Here's the low-down on when C functions need not be declared in include files.

"The include files are all screwed up!"

This is a common misconception people have when they look through the MPW C include files. People report that the declaration of a ROM or system call `foo()` has been mistakenly left out of this or that include file. Here's the low-down on when functions do not have to be declared in an include file.

The Law

A C function does not need to be declared in an include file **if** it requires glue code **and** returns a short or long integer as a result.

Routines that require glue code include:

- All routines that are marked [Not in ROM] in *Inside Macintosh*
- All register based routines (Operating System routines)
- All routines which have strings or points as arguments (and have mixed case spellings)

#165: Creating Files Inside an AppleShare Drop Folder

See also: The File Manager
AppleShare Administrator's Guide
Software Applications in a Shared Environment

Written by: Rich Andrews August 3, 1987
Modified by: Fred A. Huxham November 2, 1987
Updated: March 1, 1988

This technical note outlines the steps an application must take to create files inside AppleShare drop folders.

The AppleShare File Server allows the creation of *drop folders*. These are folders for which the user has the `Make Changes` privilege (write access), but not `See Files` (read access) or `See Folders` (search access). For an application to create a file in such a folder, the following procedure must be executed in strict order:

- Issue the `Create` call to create the new file.
- Issue a `GetCatInfo` call (if desired, to preserve the creation date).
- Set the file's creator and type (and creation date if desired), with a `SetCatInfo` call.
- Open each fork of the file with an `OpenDeny` call with `deny readers` and `deny writers` access (an attempt to open with read access will fail). If your application will need to write to both forks of the file, it must open both now.
- `Write` each fork. (Do not try to read, any attempt will fail with a privilege error.)
- `Close` each fork.

This sequence can be followed for creating any file, not just those in drop folders, so your application can always create files in this manner. There is no need to special case for drop folders.

Note that you will not be able to do a final `SetCatInfo` to set the modification date to a different value. For example, if you were making a copy of an existing file and wanted the copy to retain the original creation and modification dates, it will not be possible if the destination is in a drop folder.

#166: MPW C Functions Using Strings or Points as Arguments

See also: MPW C Manual, Appendix H:
"Functions Using Strings or Points as Parameters"

Written by: Fred A. Huxham November 2, 1987
Updated: March 1, 1988

MPW 2.0 includes new C interfaces to ROM routines which no longer do string and point conversions. These new interfaces are described here.

In MPW prior to 2.0, the C interfaces to Macintosh OS and Toolbox routines that had strings or points as arguments required following these rules:

1. Strings must be passed as C strings (null terminated).
2. Points must be passed by address.

With this method, all these functions would end up calling glue code to:

1. Convert the C strings to Pascal strings.
2. Dereference the address of the Point before pushing it onto the stack.

Because of this your applications ended up being slightly larger (due to the glue code needed) and slightly slower (due to the time it takes to execute the glue code).

MPW 2.0 C interfaces include a new set of ALL CAPS routines that have strings or points as arguments. These routines require following these rules:

1. Strings must be passed as Pascal strings (preceded by a length byte).
2. Points must be passed by value.

Calling these new routines results in smaller and faster code. In other words, these new interfaces are your friend.

Some Examples

First, some Point examples:

The routine `PtInRect`, (old style, mixed case), requires a point passed by address. The new interface:

```
pascal Boolean PTINRECT(pt,r)
    Point pt;
    Rect *r;
    extern 0xA8AD;
```

requires that the `Point` argument be passed by value.

And now, some string examples:

The routine `StringWidth`, (old style, mixed case), required a C string as an argument. The new interface:

```
pascal short STRINGWIDTH(s)
    Str255 *s;
    extern 0xA88C;
```

requires that the argument be passed as a Pascal string.

Pascal Strings

Another new feature of MPW 2.0 C is the creation of Pascal strings. You can now create a Pascal string by using the "`\p`" option. The following example demonstrates this new feature:

```
cString1 = "This is a C string"
pString2 = "\pThis is a Pascal string"
```

The first line will create a C, null terminated, string, while the second line will create a Pascal, preceded by a length byte, string.

#167: AppleShare Foreground Applications

See also: *AppleShare Administrator's Guide*

Written by: Fred A. Huxham November 2, 1987
Updated: March 1, 1988

This technical note outlines the requirements and restrictions of an AppleShare foreground application. This information pertains to AppleShare versions 1.1 and newer.

An AppleShare server requires a dedicated Macintosh. The server, however, is implemented as an interrupt-driven application that runs in the system heap of the server machine. This allows the running of a concurrent or foreground application that will live in the application heap of the server machine. An example of a foreground application is LaserShare, the LaserWriter spooler available from Apple.

An AppleShare foreground application has a few additional restrictions and requirements beyond that of a normal Macintosh application:

1. In order for AppleShare to recognize your program as a foreground application, it must contain a resource of type '`fgnd`', ID=1, containing a longword of `$00000000`.

2. Do not make any file system calls outside of server volumes' Server Folders. If a foreground application needs to create files, it is recommended that the application create a folder inside the Server Folder and then create all its files within that folder. For example, all print spooler or e-mail files must reside within the Server Folder, and preferably, within a folder that is inside the Server Folder. To find the Server Folder:

- Make a `PBHGetVInfo` call on the volume.
- Examine `ioVFndrInfo[8]` (long integer)
- If `ioVFndrInfo[8]` is non-zero, it is the directory ID of the Server Folder.

3. Do not to make file system calls or to modify the following in any way: the AppleShare server application, the Parallel Directory Structure, or the User or Group data bases within the Server Folder of any volume. Also, do not rely on the presence or formats of these structures, as they are subject to change!

4. Do not eject or unmount a volume that is not in drive 1 or 2.

5. Do not call the `Shutdown` trap; instad, quit by calling `ExitToShell` or by dropping out of the main event loop.

Macintosh
Technical Notes



Developer Technical Support

#168: HyperCard And You: Economy Edition

Revised by: jeremy j. bornstein, death dwarf of minraud
Written by: Chris Knepper

February 1991
November 1987

This Technical Note describes some HyperCard anomalies with which developers should be familiar when developing stackware, and it documents differences between HyperCard versions where appropriate.

Changes since November 1987: Consolidated Technical Notes 168, HyperCard `'snd '` Resources; 169, HyperCard 1.0.1 and 1.1 Anomalies; and 170, HyperCard File Format while adding and updating material with regard to HyperCard 2.0 and condensing or obsoleting information on bugs which have been addressed.

The 15 Billion Horsemen of the Apocalypse

With the introduction of HyperCard 2.0, many of the old bugs were quashed, and absolutely no new bugs were created. In fact, the software was so bug-free that it immediately attained Nirvana and Apple has had problems getting it to do anything since. Just kidding.

HyperCard File Format

The HyperCard file format is available for licensing on a case-by-case basis. Since HyperCard has moved to Claris, developers should contact Claris for more information if you feel that your product is a substantial and important addition to HyperCard and the Macintosh.

Since HyperCard allows developers to control data input and output to files using HyperTalk, most "needs" for file formats may be easily met by using HyperTalk and writing scripts. In particular, the following HyperTalk commands are useful:

```
open file fileName
close file fileName
read from file fileName until <delimiter>
```

Versions of HyperCard after 1.0.1 come with examples of how to use these commands. For those with HyperCard 2.0 release stacks, look in the Power Tools stack for the "Export Stack Scripts" card, which provides frameworks for building custom file I/O routines.

Import(ant) Tip

Scripts which create many new cards, e.g., scripts which import data, should be sure to keep a counter of the number of cards created and call `doMenu "Compact Stack"` every 200 or so cards. Scripts which lack this feature may cause a variety of strange results, including stack corruption, which are revealed when Compact Stack is eventually selected.

Playing Sounds

In versions prior to 2.0, HyperCard could only play format 2 `'snd '` resources. In version 2.0 and later, HyperCard supports both format 1 and format 2 `'snd '` resources, which are documented in *Inside Macintosh*, Volume V, The Sound Manager (earlier versions did not include the format 2 description).

Visual Effects On Monitors With Multiple Bit Depths

With HyperCard 2.0, visual effects work with all monitor depths up to eight bits, although these effects are faster on a one-bit depth screen. In previous versions of HyperCard, the monitor had to be set to a one-bit depth for visual effects to be visible.

High Disk Space Requirement For BackGround Printing

Running HyperCard under MultiFinder with Background Printing enabled requires extensive disk space for printing large jobs, such as printing a stack (with Print Stack... in the File menu) of 100 cards. For example, a ten card stack may only be 20K in size, but the spool file might be greater than 1MB. This requirement is due to Print Monitor's spooling the job to disk. Note that this requirement does not hold true for report printing, where no graphics need to be spooled.

Possible Printing Problems

LaserWriter Timeout Errors

The LaserWriter IINT and LaserWriter drivers prior to 6.0.1 had a bug which resulted in long jobs failing to print due to a timeout error after approximately one hour. Version 6.0.1 of the LaserWriter driver addressed this problem; however, you may still want to print large jobs in batches using a HyperTalk script.

Printing To Non-PostScript Printers

HyperCard 2.0 would not print to non-PostScript® printers. HyperCard 2.0v2 fixes this bug.

Word Wrap

HyperCard 2.0 uses TextEdit's word break routine with some optimizations. While it fixes bugs in the old HyperCard word break routine, it does things somewhat differently. Therefore, there may be significant changes in the way text fields are wrapped, and this may adversely affect the visual design of stacks intended for use with earlier versions of HyperCard. One of these changes is that HyperCard no longer considers quotation marks to be separate from their attached words.

Note that stacks designed under versions of HyperCard prior to 2.0 continue to use the old word break routine until converted to 2.0 format.

Beware The on idle Handler

An `on idle` handler in a HyperCard script executes while the user types into a field. A nasty side-effect is that the insertion point may disappear while the user types into a field, resulting in the Macintosh beeping since it cannot interpret the characters entered, or in the characters being displayed in the Message Box if it is open. A particularly insidious example is the following handler in a background script:

```
on idle
  put the time into bkgnd field "Time"
end idle
```

If the user types into a field with the example handler in a background script, the insertion point disappears from the field whenever the time changes, for example from "6:42 PM" to "6:43 PM," requiring HyperCard to change the contents of background field "Time."

A solution to work around this problem involves checking the selection within the `on idle` handler, and reselecting the selection after the `on idle` handler has done its dirty work. The following handler fixes the problem with the preceding one:

```
on idle
  put the selectedChunk into oldSelectedChunk
  put the time into bkgnd field "Time"
  select oldSelectedChunk
end idle
```

Note that if the `on idle` handler changes field contents frequently, this technique produces unsightly blinking of the selection or of the insertion point. However, if the `on idle` handler is well-behaved, it avoids such behavior and uses this technique only as insurance.

find Command

Previous versions of HyperCard had various bugs with the `find` command. HyperCard 2.0 fixes these bugs, and `find` now works with all fields, except those for which the `dontSearch` property is set to `true`. If `dontSearch` has been set to `true` for a card or a background none of the fields within the card or background are searched.

Note that when `dontSearch` is set to `true` for a background, **none** of the fields on the background or on cards which belong to that background are searched.

Optimizing find

HyperTalk's `find` command works best when at least three characters per find criteria are specified, and the stack has recently been compacted with the Compact Stack item in the File menu. Please note that specifying less than three characters or using word break characters, such as the dash (—) or space ( ), for any of these three characters results in a slower find.

The more trigrams that are specified for the `find` (strings with at least three characters), the faster the find; thus, the following `find` in a company phone list:

```
find "Thanasis Metagreek Appraisal"
```

would find "Thanasis Kehagias" in "Metagreek Appraisal Department" faster than would:

```
find "Th"
find "Thanasis"
find "Thanasis K"
find "Thanasis Kehagias"
```

Paint Tools and HyperTalk

HyperTalk can control all painting tools and commands except the polygon tool and the special effects tools (Rotate, Distort, Slant, and Perspective).

Selecting Contents Of A Field With HyperTalk

There are several ways of selecting text in a field with HyperTalk: by simulating a double click, a Shift-click, or a drag. The three button scripts presented here show how to do this.

The first shows a method of simulating a double click. This script performs a double click at the `loc` of a card field (approximately the center of the field's rectangle) then puts the selection into another card field:



Figure 1–Simulating A Double Click

```
on mouseUp -- card button "Double Click"
  click at the loc of field "myField1"
  click at the loc of field "myField1"
  if the selection is empty
  then put "No selection" into card field "DoubleClickSelection"
  else put the selection into card field "DoubleClickSelection"
end mouseUp
```

This second script demonstrates a method of selecting text in a card field with the shift-click and puts the selection in another card field:



Figure 2–Simulating A Shift-Click

```
on mouseUp -- card button "Shift Click"
  get the rect of bkgnd field "myField2"
  click at item 3 to 4 of it
  click at item 1 to 2 of it with ShiftKey
  if the selection is empty
  then put "No selection" into card field "ShiftClickSelection"
  else put the selection into card field "ShiftClickSelection"
end mouseUp
```

The third script presents a method of selecting text by dragging over it. This method, however, requires `with shiftKey` as a parameter to `drag` in order to work correctly:



Figure 3–Simulating A Drag

```
on mouseUp -- card button "Drag Shift"
  get the rect of bkgnd field "myField3"
  drag from item 1 to 2 of it to item 3 to 4 of it with shiftKey
  if the selection is empty
  then put "No selection" into card field "DragShiftSelection"
  else put the selection into card field "DragShiftSelection"
end mouseUp
```

Dialing The Telephone

dial Command

System Software 6.0.7 has a bug on the Macintosh IIsi which causes HyperCard 2.0 to run the tones together when executing a `dial` command. Version 2.0v2 fixes this bug; however, it contains another bug in which a comma (,) in the `dial` command is ignored if it is preceded by more than three characters which produce tones.

HyperCard 2.0v2 fixes all known bugs with the dial command other than the one just documented. In addition, the dial command now supports the A, B, C, and D buttons found on military telephones.

.TouchTone 'DRVR'

Note that the .TouchTone `'DRVR'` was removed in HyperCard 1.1.

Launching Applications Under MultiFinder With open

HyperCard 2.0 returns a value in `the result` if an application launch with the `open` command fails; this message is either "out of memory" or "couldn't open that application," depending upon the reason it could not launch the given application.

mouseDown and mouseUp Messages in Scrolling Fields

HyperCard does not send the `mouseDown` and `mouseUp` messages to scrolling fields when it detects a mouse click in the field's scroll bar.

Passing it As A Parameter To Handlers

Passing `it` as a parameter to a handler is no different than passing any other parameter to a handler. This means that: it is always a local variable to the handler in which it is used, and it is passed to a handler by value, not by reference.

Disappearing Resource Forks

If an 'XCMD' adds a resource fork to a currently open stack, HyperCard does not know about it. When a stack is compacted, and HyperCard does not think that the stack has a resource fork, then the resource fork is not preserved after the compaction. The solution to work around this is to `go home` before compacting the stack. Then, when the stack is opened again, HyperCard recognizes that the stack contains a resource fork and behaves appropriately.

Using Globals From 'XCMD' Resources

It used to be the case that one could not create a global from an 'XCMD'. HyperCard 2.0 fixed this bug; however, there is an issue in that messages an 'XCMD' passes back to HyperTalk are executed in the scope of the handler which called the 'XCMD'. This situation means that the calling handler must declare `global someGlobal` for the following to work:

```
SendCardMessage(paramPtr,"\p put someGlobal into cd fld 1");
```

Some people may notice that there is still the problem of how to make sure the handler declares the global if you do not want the 'XCMD' to be dependent upon its calling script. Simply replace the example with the following code:

```
SendCardMessage(paramPtr,"\p global someGlobal \n put someGlobal into cd fld 1");
```

Note the return character (\n) in the second example above—the message passed to HyperCard becomes a two-line message (yes, it is possible). The first line makes the handler's scope aware of the global `someGlobal`, while the second line does what you want with the global. It is now safe to pass a global variable back to HyperTalk since the 'XCMD' has added it to the local scope of the handler.

La Bomba (Ya No Soy Marinero)

It is imperative that scripts test the global property `the heapspace` in a variety of scripting situations which reduce the amount of available heap space. HyperTalk needs to have about 32K of heap space available at all times when a script is running. If `the heapspace` drops below this value, HyperCard aborts the script with an "out of memory" error. Furthermore, in its attempts to clean up, HyperCard may empty those global variables upon which it was working when the error occurred.

Unfortunately, there is no way for a script to trap for out of memory errors when they occur, although it is sometimes possible to enter the debugger to determine exactly what was happening when memory ran out like so many gamma particles from Einsteinium. However, scripts can check `the heapspace` whenever they are about to do something which might require much memory, then either fall back to a less memory-intensive way of performing the same operation or simply report the error and abort safely.

For example, in scripts which import large amounts of data into a variable, test `the heapspace` frequently, and when it gets small, dump the contents of the variable into a field.

Another good use for `the heapspace` is in testing whether or not `'XCMD'` or `'XFCN'` resources are properly disposing of memory. A test script can call the external command repeatedly, displaying `the heapspace` in the Message Box each time. For example, the following script repeatedly calls the `'XCMD'` flash, putting `the heapspace` into the Message Box after each call, until the Shift key is pressed. If the value displayed in the Message Box decreases consistently, it should be clear that the `'XCMD'` is eating memory, perhaps by allocating blocks with calls to `_NewHandle` which are not disposed of with a corresponding call to `_DisposHandle`.

```
on mouseUp
  repeat while the shiftKey is up -- depress the Shift Key to stop
    flash 2 -- call your XCMD or XFCN
    put the heapspace -- monitor this value to see if it's decreasing
  end repeat
end mouseUp
```

Further Reference:

- *HyperCard Script Language Guide*
- *Getting Started With HyperCard 2.0*
- HyperCard 2.0 Release Notes
- *The Age of Reason*

PostScript is a registered trademark of Adobe Systems Incorporated.



#169: HyperCard 1.0.1 And 1.1 Anomalies

Revised by: jeremy j. bornstein, death dwarf of minraud
Written by: Chris Knepper

February 1991
November 1987

This Technical Note formerly described some HyperCard anomalies between HyperCard 1.0.1 and 1.1.

Changes since March 1988: Merged contents into Technical Note #168, HyperCard And You: Economy Edition.

This Note formerly described some HyperCard anomalies between versions 1.0.1 and 1.1. This information has been updated for HyperCard 2.0 and integrated into Technical Note #168, HyperCard And You: Economy Edition.



#170: HyperCard File Format

Revised by: jeremy j. bornstein, death dwarf of minraud
Written by: Chris Knepper

February 1991
November 1987

This Technical Note formerly discussed the proprietary nature of the HyperCard file format and Apple's policy not to license it.
Changes since March 1988: Merged contents into Technical Note #168, HyperCard And You: Economy Edition.

This Note formerly discussed the proprietary nature of the HyperCard file format and Apple's policy not to license it. This information has been updated for HyperCard 2.0 under Claris and integrated into Technical Note #168, HyperCard And You: Economy Edition.

Macintosh
Technical Notes

Developer Technical Support

#171: Things You Wanted to Know About _PackBits*
*But Were Afraid to Ask

Revised by: Guillermo Ortiz & Jon Zap — April 1990
Written by: Cameron Birse — November 1987

This Technical Note describes the format of data packed by the Toolbox utility `_PackBits` and documents a change to the `srcBytes` limit and possible worst case. Although you can simply unpack this data using `_UnPackBits`, Apple provides this information for the terminally curious and for those manipulating MacPaint® documents or PICT files by hand. **Warning:** This format information is subject to change.
Changes since February 1989: Added information about the new `srcBytes` limit and worst case results and clarified the description of how to decode packed bytes.

Length Doesn't Matter

Inside Macintosh, Volume I-470, The Toolbox Utilities, describes the Pascal interface to the `_PackBits` trap as follows:

```
PROCEDURE PackBits(VAR srcPtr,dstPtr:Ptr; srcBytes:INTEGER);
```

The accompanying text states that `srcBytes`, the length of your uncompressed data, should not be greater than 127, and that in the worst case, the compressed data can be `srcBytes + 1`. To pack more than 127 bytes, you had to break the data up into 127-byte groups and call `_PackBits` on each group. Beginning with System Software 6.0.2, this limit of 127 bytes is no longer valid. The new limit is 32,767 bytes, which is the maximum positive number that `srcBytes` can hold. The worst case can be determined according to the following formula:

```
(srcBytes + (srcBytes+126) DIV 127)
```

which is comparable to what you would get if you broke up the data into 127-byte groups and picked up an additional byte for each group.

Mommy, How Do They Make Packed Bits?

The first byte is a flag-counter byte that specifies whether or not the the following data is packed, and the number of bytes involved. If this first byte is a negative number, then the following data is packed. In this case, the number is the two's complement of a zero-based count of the number of times the data byte repeats when expanded. There is one data byte following this first byte in packed data. The byte after the data byte is the next flag-counter byte.

If the flag-counter byte is a positive number, then the following data is unpacked. In this case, the number is a zero-based count of the number of incompressible data bytes that follow. There are

(flag-counter+1) data bytes following the flag-counter byte. The byte after the last data byte is the next flag-counter byte.

Note that there is no way to know, given a pointer to the start of packed data, when you have reached the end of the packed data. This is why you need to know either the length of the packed or unpacked data before you start unpacking. _UnPackBits requires the length of the unpacked data.

Consider the following example:

Unpacked data:

```
AA AA AA 80 00 2A AA AA AA AA 80 00 2A 22 AA AA AA AA AA AA AA AA AA AA
```

After being packed by _PackBits:

```
FE AA                   ; (-(-2)+1) = 3 bytes of the pattern $AA
02 80 00 2A             ; (2)+1 = 3 bytes of discrete data
FD AA                   ; (-(-3)+1) = 4 bytes of the pattern $AA
03 80 00 2A 22          ; (3)+1 = 4 bytes of discrete data
F7 AA                   ; (-(-9)+1) = 10 bytes of the pattern $AA
```

or

```
FE AA 02 80 00 2A FD AA 03 80 00 2A 22 F7 AA
*     *           *     *              *
```

The bytes with the asterisk (*) under them are the flag-counter bytes. _PackBits only packs the data when there are three or more consecutive bytes with the same data, otherwise it just copies the data byte for byte (and adds the count byte).

Note: The data associated with some PICT opcodes, $0098 (PackBitsRect) and $0099 (PackBitsRgn), contain PixData which is basically made of _PackBits data. It should be noted, though, that the format for PixData includes a byteCount or length in addition to the data described in this Note.

For example, the following is the result of decoding a sample PICT2:

```
data 'PICT' (25534) {
    0936 0000 0000 0007 001E              /* pic size, picFrame */
    0011 02FF                             /* pict2              */
    0C00                                  /* header             */
         FFFF FFFF 0000 0000 0000 0000 001E 0000 0007 0000 0000 0000
    001E                                  /* def hilite         */
    0001                                  /* clipRgn            */
         000A 0000 0000 0007 001E
    0098                                  /* PackBitsRect       */
         801E                             /* rowbytes of 30     */
         0000 0000 0007 001E              /* Bounds             */
         0000                             /* packType           */
         0000                             /* version            */
         0000 0000                        /* packSize           */
         0048 0000                        /* hRes               */
         0048 0000                        /* vRes               */
         0000                             /* pixelType          */
         0008                             /* pixelSize          */
         0001                             /* cmpCount           */
         0008                             /* cmpSize            */
         0000 0000                        /* planeBytes         */
```

```
 0000 1F10                          /* pmTable             */
 0000 0000                          /* pmReserved          */
/*color table*/
      0000 4CBC                     /* ctSeed              */
      8000                          /* ctFlags             */
      00FF                          /* ctSize              */
           0000 FFFF FFFF FFFF
           ...                  /* 254 ColorSpec's omitted */
           0000 0000 0000 0000
 0000 0000 0007 001E                /* srcRect             */
 0000 0000 0007 001E                /* dstRect             */
 0000                               /* srcCopy             */

 /* Now we have the scan line data packed as follows:
    [bytecount for current scan line] [data as defined above]
    If rowBytes is > 250 then byteCount is a word else is a byte
    (in this case, byteCount is a byte)
    note that each unpacked row adds to 30 rowBytes
 */

 /* line 1, byte count is 2 (best case for a row)   */
 02
    E3 FF                /* -(-29) + 1 = 30 FF's    */
 /* line 2, byte count is 19 (0x13)                 */
 13
    01 FF 23             /* 1+1 data bytes          */
    FE 00                /* -(-2)+1 0's             */
    FC 23                /* -(-4)+1 0x23's          */
    FE 00                /* 3 0's                   */
    FC 23                /* 5 0x23's                */
    FE 00                /* 3 0's                   */
    FC 23                /* 5 0x23's                */
    FE 00                /* 3 0's                   */
    00 FF                /* 1 data byte             */
 /* line 3, byte count is 28                        */
 1C
    02 FF 00 23          /* 3 data bytes            */
    FE 00                /* 3 0's                   */
    FE 23                /* 3 0x23's                */
    01 00 23             /* 2 data bytes            */
    FE 00                /* 3 0's                   */
    FE 23                /* 3 0x23's                */
    01 00 23             /* 2 data bytes            */
    FE 00                /* 3 0's                   */
    FE 23                /* 3 0x23's                */
    04 00 23 00 00 FF    /* 5 data bytes            */
 /* line 4, byte count is 31 (worst case for a row) */
 1F
    03 FF 00 00 23       /* 4 data bytes            */
    FE 00                /* 3 0's                   */
    00 23                /* 1 data byte             */
    FE 00                /* 3 0's                   */
    00 23                /* 1 data byte             */
    FE 00                /* 3 0's                   */
    00 23                /* 1 data byte             */
    FE 00                /* 3 0's                   */
    00 23                /* 1 data byte             */
    FE 00                /* 3 0's                   */
    00 23                /* 1 data byte             */
    FE 00                /* 3 0's                   */
    00 23                /* 1 data byte             */
    FE 00                /* 3 0's                   */
    02 23 00 FF          /* 3 data bytes            */
```

```
        /* line 5, byte count is 28                        */
        1C
           01 FF 00             /* 2 data bytes            */
           FE 23                /* 3 0x23's                */
           01 00 23             /* 2 data bytes            */
           FE 00                /* 3 0's                   */
           FE 23                /* 3 0x23's                */
           01 00 23             /* 2 data bytes            */
           FE 00                /* 3 0's                   */
           FE 23                /* 3 0x23's                */
           01 00 23             /* 2 data bytes            */
           FE 00                /* 3 0's                   */
           FE 23                /* 3 0x23's                */
           00 FF                /* 1 data byte             */
        /* line 6, byte count is 18                        */
        12
           00 FF                /* 1 data byte             */
           FC 23                /* 5 0x23's                */
           FE 00                /* 3 0's                   */
           FC 23                /* 5 0x23's                */
           FE 00                /* 3 0's                   */
           FC 23                /* 5 0x23's                */
           FE 00                /* 3 0's                   */
           FD 23                /* 4 0x23's                */
           00 FF                /* 1 data byte             */
        /* line 7, byte count is 2 (best case for a row)   */
        02
           E3 FF                /* 30 0xFF's               */
        00  /* pad so next command starts at word boundary */

    00FF                        /*end of pic               */
};
```

Further Reference:

- *Inside Macintosh*, Volume I-465, The Toolbox Utilities
- *Inside Macintosh*, Volume V-39, Color QuickDraw
- Technical Note #86, MacPaint Document Format

MacPaint is a registered trademark of CLARIS Corporation

#172: Parameters for MDEF Message #3

See also: The Menu Manager

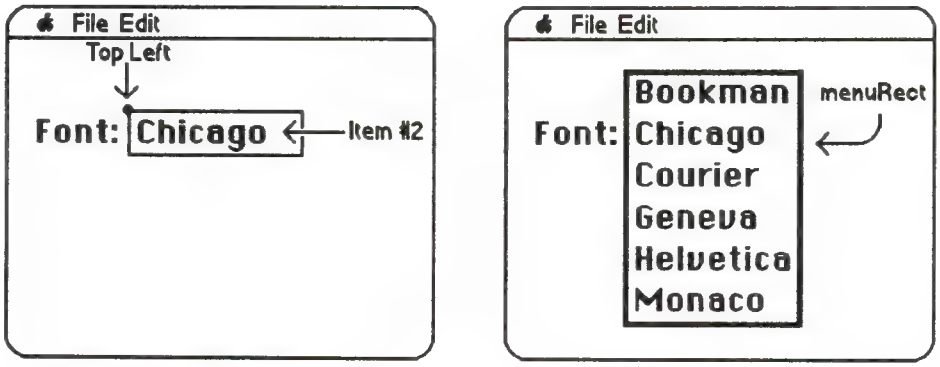
Written by: Chris Derossi November 2, 1987
Updated: March 1, 1988

In order to support popup menus, menu definition procedures (MDEFs) must now respond to a new message, `mPopupMsg`. `mPopupMsg` is message number 3. When your MDEF is called with this message, it should calculate the rectangle in which the popup menu should appear.

The interface to an `MDEF` is

```
PROCEDURE MyMDEF(message: Integer; theMenu: MenuHandle; VAR menuRect:
                 Rect; hitPt: Point; VAR whichItem: Integer);
```

For `mPopupMsg`, the `message` parameter will be 3 and `theMenu` will be a `MenuHandle` to your menu. The MDEF should compute a rectangle for the menu such that the item passed in `whichItem` will be displayed at `hitPt`. See the figure below:



The `hitPt` parameter, though, is NOT a `Point`. Instead, this parameter is used to pass the top left of the item, passing the `top` coordinate and then the `left` coordinate. This is the opposite order of the fields in a `Point`. The values can be used together as a `LongInt`, with `left` in the high word and `top` in the low word, or separately as two `Integers`.

A more correct Pascal interface to the MDEF (for the `mPopupMsg` **only**) would be:

```
PROCEDURE MyMDEF(message: Integer; theMenu: MenuHandle; VAR menuRect:
                 Rect; top, left: Integer; VAR whichItem: Integer);
```

Note: The MPW interface files incorrectly list `mPopupMsg` as 4; it should be 3.

#173: PrGeneral Bug

See also: The Print Manager
Technical Note #128—PrGeneral

Written by: Scott "ZZ" Zimmerman November 2, 1987
Updated: March 1, 1988

This technical note documents a bug in the implementation of the PrGeneral procedure in the LaserWriter driver version 4.0. The bug has to do with the format of the information returned by the GetRslData opcode. This technical note will also describe a workaround for the problem.

One of the opcodes supported by the `PrGeneral` procedure (Technical Note #128) is named `GetRslData`. The `GetRslData` operation initializes a resolution record that is of the following form:

```
TRslRg = RECORD {used in TGetRslBlk}
     iMin: Integer;     {0 if printer only supports discrete resolutions}
     iMax: Integer;     {0 if printer only supports discrete resolutions}
END;

TRslRec = RECORD {used in TGetRslBlk}
     iXRsl:       Integer;     {a discrete, physical X resolution}
     iYRsl:       Integer;     {a discrete, physical Y resolution}
END;

TGetRslBlk = RECORD {data block for GetRslData call}
     iOpCode:     Integer;     {input; = getRslDataOp}
     iError:      Integer;     {output}
     lReserved:   LongInt;     {reserved for future use}
     iRgType:     Integer;     {output; this declaration is for RgType1}
     XRslRg:      TRslRg;      {output; range of X resolutions}
     YRslRg:      TRslRg;      {output; range of Y resolutions}
     iRslRecCnt:  Integer;     {output; how many RslRecs follow}
     rgRslRec:    ARRAY[1..27]
                  OF TRslRec; {output; number used depends on printer type}
END;
```

The LaserWriter 4.0 implementation has a bug that affects the `YRslRg` and `XRslRg` fields of the `TGetRslBlk` record. The correct values for the fields are:

```
TGetRslBlk.XRslRg.iMin := 25;
TGetRslBlk.XRslRg.iMax := 1500;
TGetRslBlk.YRslRg.iMin := 25;
TGetRslBlk.YRslRg.iMax := 1500;
```

Unfortunately, the information returned by the LaserWriter 4.0 version of `PrGeneral` is:

```
TGetRslBlk.XRslRg.iMin := 25;
TGetRslBlk.XRslRg.iMax := 25;
TGetRslBlk.YRslRg.iMin := 1500;
TGetRslBlk.YRslRg.iMax := 1500;
```

The recommended workaround for this problem is to use the `PrDrvrVers` function (*Inside Macintosh* II-163) to find out which version of the print driver you are using. If you are using 4.0, modify the resolution data before using it. The following code fragment illustrates this workaround:

```
PROCEDURE CheckRslRecord(VAR theRslRecord: TGetRslBlk);
CONST
     BogusDriver = 40;
BEGIN
     IF PrDrvrVers = BogusDriver THEN BEGIN
          theRslRecord.XRslRg.iMax := theRslRecord.YRslRg.iMax;
          theRslRecord.YRslRg.iMin := theRslRecord.XRslRg.iMin;
     END;
END;
```

When the bug is fixed in a future version of the driver, the `CheckRslRecord` procedure will no longer have any effect on the resolution record. This will make sure your application gets the correct resolution data no matter which version of the driver is being used.

#174: Accessing the Script Manager Print Action Routine

See also: The Script Manager

Written by: Mark Davis November 2, 1987
Updated: March 1, 1988

This technical note describes how Print Drivers can access the Script Manager Print Action routine to print unconventional text, such as Japanese or Arabic.

General Notes

Scripts such as Japanese or Arabic modify the normal QuickDraw text handling in order to represent text properly. On the screen, this is done by trapping `StdText` and `StdTxtMeasure`, and transforming the text before printing. For example, for Hebrew or Arabic the text might be reversed, since text normally goes from right to left in those scripts.

Print drivers require slightly different handling, for two reasons:

1. A print driver might not call the standard QuickDraw procedures. For example, the LaserWriter writes directly in PostScript instead.

2. A print driver might need to format the text, for accurate line-layout. In this case, the text needs to be transformed before the driver performs line-layout. If the driver is spooling the text, and will replay the text a second time, the text cannot be transformed a second time, since that would ruin the appearance.

For example, the ImageWriter driver calls QuickDraw procedures twice, once to spool and once to unwind the spooling. The text must be transformed when spooling, so that line layout can be done, but when unwinding, the transformation must be turned off completely.

Note that some drivers, such as the LaserWriter, use QuickDraw re-entrantly: the application program calls a QuickDraw routine, which is directed to the driver's `grafProcs`, which in turn call QuickDraw internally to put up status messages on the screen. The Print Action procedure handles the text properly so that the text transformations are enabled during the re-entrant calls, so that the status messages will be properly formatted.

When To Call the Print Action Routine

The Script Manager Print Action routine allows the print driver to be independent of the particular scripts being used. The printing driver should call this routine whenever it changes the `grafProcs` in the printing `grafPort`. The Print Action routine will then substitute `grafProcs` of its own in the `grafProcs` record, saving the original routine addresses.

The Print Action routine will actually call a Print Action routine for each script system that is currently installed. Each of the script Print Action routines will do the appropriate tasks for its system.

Calling the Print Action Routine

To call the Print Action routine, the driver should use the following code:

```
intlGlobals         equ         $ba0              ; international globals
printActionOff      equ         $16               ; offset to PrintAction proc ptr

; get procedure pointer to call

     tst.w          Rom85                         ; on a Macintosh + or better?
     blt.s          @PrintActionDone              ; no, skip
     move.l         intlGlobals,d2                ; get international globals
     ble.s          @PrintActionDone              ; not there, skip
     move.l         d2,a0                         ; in address register
     move.l         printActionOff(a0),d2         ; get print action address
     beq.s          @PrintActionDone              ; not there, skip
     move.l         d2,a0                         ; in address register

; set up arguments to call

     move.l         <myPort>,d0                   ; pass the port
     move.w         <myVerb>,d1                   ; pass the verb
     jsr            (a0)                          ; call the procedure
@PrintActionDone
```

Print Action Routine Verbs

There are currently three verbs to pass to the Print Action routine.

```
     paUnwindText              equ               -1
     paSpoolText               equ               1
     paNoQuickDraw             equ               3
```

Use the `paUnwind` verb to ensure that the text is not transformed before your `StdText` procedure receives the text. This verb is used when playing back stored text that has already been transformed.

The other two verbs (`paNoQuickDraw` and `paSpoolText`) are used to ensure that the text is transformed before your `StdText` procedure receives the text. The `paSpoolText` verb is used when your driver will use QuickDraw to image the text in the printing `grafPort`. The `paNoQuickDraw` verb is used when the text is not drawn into the printing port by going through QuickDraw (e.g. the LaserWriter). In that case some languages (e.g. Japanese) which use an extended font structure may need to recast the text calls as `CopyBits` calls.

As mentioned above, some applications may call QuickDraw from within the driver, as when a status window is updated. During any `StdTxtMeasure` calls in the driver during the application's call to `StdText`, the port is checked against the printer port. If they match, then the text is not transformed. Otherwise, the text is transformed.

The solutions adopted by the Print Action routine assume that the print driver does not measure or draw text except within calls to `StdTxtMeasure` or `StdText`. If your driver does text buffering (as for line layout), make sure that any measurements are performed within these two calls. For example, you might buffer both the text and its screen width as measured by QuickDraw.

#175: SetLineWidth Revealed

See also: *LaserWriter Reference Manual*
PostScript Language Reference Manual
PostScript Language Tutorial and Cookbook

Written by: Scott "ZZ" Zimmerman November 2, 1987
Updated: March 1, 1988

This technical note describes the internal implementation, and correct method of using, the `SetLineWidth` Picture Comment.

The `SetLineWidth` picture comment provides a way of accessing PostScript's 'setlinewidth' operator. Since the LaserWriter resolution is roughly four times that of the Macintosh screen, fractional line widths can be printed. The `SetLineWidth` PicComment provides a way for applications to access these fractional line widths through PostScript, without having to use floating point numbers.

First of all, the LaserWriter has an internal state that is stored in a number of PostScript variables. For more information on PostScript variables, see the *PostScript Language Reference Manual*. Some operations performed on the LaserWriter cause the values of these variables to change. One of these variables contains the width of the printer's pen. The `SetLineWidth` picture comment works by changing the value of this variable.

Before we look at what the `SetLineWidth` comment does, let's look at the argument passed to the comment. The argument is represented as a QuickDraw Point, however it is interpreted by the LaserWriter as a fraction. The LaserWriter interprets a `point(h,v)` to be a real number whose value is `(v / h)`. This means that a point whose value is h=2, v=1, will be converted to 0.5 before being used by the LaserWriter. If you wanted to pass a value of 0.25, you would pass a point whose value is h=4, v=1. For 1.25, pass a point, h=4, v= 5.

In addition to the pen width variable, there is a variable that is used for scaling the pen's width. This variable, named `pnm` for PeN Multiplier, contains a real number which is applied to the pen width. The default value of `pnm` is 1.0, which causes no scaling of the line width.

Whenever the `SetLineWidth` PicComment is sent to the LaserWriter, the current value of `pnm` is replaced by the value passed to the PicComment. The current pen size is then scaled by the new value of `pnm`. The following example will display four lines of different sizes. It is meant to illustrate the interaction between the QuickDraw `PenSize` procedure and the `SetLineWidth` PicComment.

```
TYPE
     widthHdl     = ^widthPtr;
     widthPtr     = ^widthPt;
     widthPt      = Point;

VAR
    theWidth:    widthHdl;

BEGIN

     (* Initialize the print manager as per Inside Macintosh II-155. *)
```

At this point, it is assumed that `PrPageOpen` has been called, and the print manager is ready to accept data.

The first thing we do is set the scaling factor to 1.0. This way, no scaling will be performed when we call `PenSize`.

```
     theWidth := widthHdl(NewHandle(SizeOf(widthPt)));
     (*Real programs do error checking here... *)
     SetPt(theWidth^^, 1, 1);
     PicComment(SetLineWidth, SIZEOF(widthPt), Handle(theWidth));
```

Here we call `PenSize`. Because the `pnm` has been set to 1.0, the pen size(1,1) times the multiplier (1.0) yields 1,1.

```
     PenSize(1, 1);
     MoveTo(50, 100);
     LineTo(500, 100);
     MoveTo(50, 125);
     DrawString('1 point thickness.');
```

Now we will use the `SetLineWidth` PicComment to change the pen size. Note that when we change the scaling factor, the pen size changes as well.

```
     SetPt(theWidth^^, 1, 5);
     PicComment(SetLineWidth, SIZEOF(widthPt), Handle(theWidth));
     MoveTo(50, 200);
     LineTo(500, 200);
     MoveTo(50, 225);
     DrawString('5.0 times 1 point pen size = 5 point thickness.');
```

If any calls to `PenSize` are made at this point, the new pen size will be scaled by 5.0. This is because the `SetLineWidth` PicComment is still in effect. We will now send a `SetLineWidth` PicComment to revert the scaling factor back to 1.0.

```
    SetPt(theWidth^^, 5, 1);
    PicComment(SetLineWidth, SIZEOF(widthPt), Handle(theWidth));
    MoveTo(50, 300);
    LineTo(500,300);
    MoveTo(50, 325);
    DrawString('0.2 times 5 point pen size = 1 point thickness.');
```

Since the scaling is once again 1.0, `PenSize` calls at this point will not be scaled. Here we explicitly set the scaling factor to 1.0 before changing the pen size. This makes it easier to see what scaling will be applied to the next call to `PenSize`.

```
SetPt(theWidth^^, 1, 1);
PicComment(SetLineWidth, SIZEOF(widthPt), Handle(theWidth));
PenSize(1, 1);
MoveTo(50, 400);
LineTo(500,400);
MoveTo(50, 425);
DrawString('1.0 times 1 point pen size = 1 point thickness');
(* Dispose of the handle when you are through with it! *)
DisposHandle(Handle(theWidth));
```

When printed, the above example will produce the following:

---

1 point thickness.

5.0 times 1 point pen size = 5 point thickness.

0.2 times 5 point pen size = 1 point thickness.

1.0 times 1 point pen size = 1 point thickness.

To summarize, there are four things to remember when using the `SetLineWidth` PicComment:

1. The argument to the `SetLineWidth` PicComment is specified as a point, though it is actually interpreted by the LaserWriter as a real number. The point value is specified as `h,v`, and the LaserWriter interprets the value as `v / h`.

2. The `SetLineWidth` PicComment affects both the height and width of the pen, even though the name suggests otherwise.

3. When you send the `SetLineWidth` PicComment, the current pen size will be scaled. Any drawing that is done after the PicComment is set, will be done with the scaled pen size.

4. When you call the QuickDraw `PenSize` procedure, the pen size will be scaled after it has been set. For example, if your scaling factor is 0.5, and you set the pen size to 2,2, the actual pen size will be 1,1. If you don't want the scaling to occur, make sure to send a `SetLineWidth` PicComment, with the point argument set to 1,1. The next call to `PenSize` will then be scaled by 1.0, which will have no effect.



#176: Macintosh Memory Configurations

Revised by: Jack Robson — April 1992
Prior Revisions: Mark Baumwell, Craig Prouse, and Dennis Hescox
Written by: Cameron Birse — November 1987

This Technical Note describes the different possible memory configurations of all models of the Macintosh family that use Single In-line Memory Modules (SIMMs) as well as the non-SIMM memory upgrade options of the Macintosh Portable and Macintosh Classic. (Special thanks to Brian Howard for the Macintosh Plus and original SE drawings, and for the inspiration for the other drawings.) This Note also describes the obstacles to using four megabit (Mbit) DRAM SIMMs in Apple Macintosh products to date.

Changes since November 1991: Corrected error on the RAM configuration chart (page 2); additional information added to Quadra 900 section (page 15).

Developer Technical Support receives numerous questions about the many different possible configurations of RAM on the different Macintosh models, so we'll attempt to answer these questions in this Technical Note, as well as to provide a showcase for some outstanding Macintosh Plus and SE artwork by Apple engineer Brian Howard. Interested readers should refer to the *Guide to the Macintosh Family Hardware*, Second Edition, which contains much more detail on the memory configurations and specifications for all Macintosh models released to date. For information on the newer Macintosh models not mentioned in the *Guide to the Macintosh Family Hardware*, please refer to the companion developer notes for those particular products.

RAM Configuration Chart

Caveat: The upper physical RAM totals expressed here assume the use and compatibility of 4 and 16 MB SIMMs. Since Apple has not yet thoroughly tested SIMMs larger than 1 MB with our Macintosh line, these upper limits should be considered theoretical. At this point Apple cannot claim that these SIMM sizes will work, nor can we guarantee any information in this Tech Note that pertains to the use of 4 and 16 MB SIMMs (read: use them at your own risk).
All numbers are expressed in terms of megabytes (MB) unless otherwise noted.

	Permanent RAM	No. of SIMM slots	Allowable SIMM sizes	Physical RAM Totals	Req. Speed	TN Page
Plus	0	4	256K,1	512K,1,2,2.5,4	150 ns	3,4
SE	0	4	256K,1	512K,1,2,2.5,4	150 ns	3,5,6
Classic	1	2*	256K,1	1,2,2.5,4	150 ns	3,8
Classic II‡	2	2	1,2,4	2,4,6,10	100 ns	17
SE/30	0	8	256K,1,4,16	1,2,4,5,8…128†	120 ns	7,9
II	0	8	256K,1,4,16††	1,2,4,5,8…68†	120 ns	7,9
IIx	0	8	256K,1,4,16	1,2,4,5,8…128†	120 ns	7,9
IIcx	0	8	256K,1,4,16	1,2,4,5,8…128†	120 ns	7,12
LC‡	2	2	1,2,4,16	2,4,6,10	100 ns	8,10
IIsi‡	1	4	256K,512K,1,2,4,16	1,2,3,5,9,17…65	100 ns	8,10
IIci‡	0	8	256K,1,4,16	1,2,4,5,8,16,		
				10,17,20,32…128	80 ns	10,12
Portable	1	0**	n/a	1,2,3,4,5,6,7,8,9***	100 ns	11,12
Portable (backlit)	1	0**	n/a	1,2,3,4,5,6,7,8***	100 ns	
IIfx‡	0	8	1,4,16	4,8,16,20,32…128	80 ns	13,14
Quadra 700‡	4	4	1,4,16	4,8,20…68	80 ns	16
Quadra 900‡	0	16	1,4,16	4,8,12,16,20,		
				24,28,32,36,		
				40,48,52,64…256	80 ns	15
PowerBook 100	2	0**	n/a	2,4,6,8	n/a	19
PowerBook 140‡	2	0**	n/a	2,4,6,8	n/a	18
PowerBook 170‡	2	0**	n/a	2,4,6,8	n/a	18

*The Macintosh Classic has 1 MB of RAM soldered onto the motherboard. Additional RAM can be added by using an expansion card. Apple Macintosh Classic 1 MB Memory Expansion Card has 1 MB of additional RAM and two SIMM connectors.

**The Macintosh Portable and the PowerBook computers allow you to add RAM by using an expansion card. These expansion cards can have from 1 MB to 4 MB of memory for the Portable, 1 MB to 3 MB for the backlit Portable, and 2, 4, or 6 MB for the PowerBook line.

***If the PDS slot is used for other peripherals, then the maximum amount of RAM (by using a RAM expansion card) is 5 MB for the Macintosh Portable, and 4 MB for the backlit Macintosh Portable.

‡These systems have ROMs that are capable of 32-bit addressing (when using the appropriate system software, such as System 7 or A/UX).

†The Macintosh II, IIx, IIcx, and SE/30 can benefit from larger SIMM sizes and address more than 8 MB of RAM by using either A/UX or the 32-bit addressing software solution called MODE32™ in conjunction with System 7. This will allow you to address up to 128 MB on the IIx, IIcx, and SE/30, and up to 68 MB on the Macintosh II (four 1 MB SIMMs in Bank A, four 16 MB SIMMs in Bank B). If you use SIMMs larger than 1 MB on the Macintosh II or IIx, you must have a PMMU and special SIMMs with PAL™ logic on them. Please refer to pages 7 and 20 of this Tech Note for more information on these SIMMs. MODE32, by Connectix, has been made available at no charge to all Apple customers. For more information about MODE32, please contact Apple at 1-800-776-2333.

††SIMMs greater than 1 MB can only be in SIMM Bank B. Please refer to Page 7 for more Macintosh II information.

Warning: Because the video monitor is built in, there are dangerous voltages inside the cases of the Macintosh Plus, SE, Classic, Classic II, and SE/30 computers. The video tube and video circuitry may hold dangerous charges long after the computer's power is turned off. Opening the case of these computers requires special tools and may invalidate your warranty. Installation of RAM in the SIMM sockets in these computers should be done by qualified service personnel only.

Macintosh Plus

The Macintosh Plus has the following possible configurations (see Figure 1):

512K, using two 256 Kbit SIMMs
1 MB, using four 256 Kbit SIMMs
2 MB, using two 1 Mbit SIMMs
2.5 MB, using two 1 Mbit SIMMs and two 256 Kbit SIMMs
4 MB, using four 1 Mbit SIMMs

It is important to place the SIMMs in the correct location when using a combination of SIMM sizes, as in the 2.5 MB example, and to make sure the right resistors are cut. Refer to Figure 1 for the correct location of the SIMMs and size resistors.

Macintosh SE

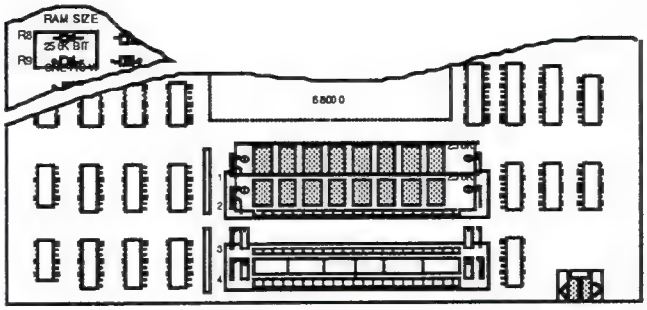
The Macintosh SE configurations (the original motherboard as well as the revised motherboard with a memory jumper selector) are the same as the Macintosh Plus, except physical locations on the motherboard are different. In addition, memory configurations with only two SIMMs (for example, 512K and 2 MB) use slots 3 and 4 on the revised SE motherboard instead of slots 1 and 2 like the original motherboard and Macintosh Plus. Refer to Figures 2 and 3 for the correct locations and settings.

Macintosh Classic

The Macintosh Classic has the following possible configurations (see Figure 4):

1 MB, using eight 128 Kbit DRAMs soldered to the motherboard
2 MB, using the memory expansion card and setting the jumper to "SIMM NOT INSTALLED"
2.5 MB, using two 256 Kbit SIMMs on the memory expansion card and setting the jumper to "SIMM INSTALLED"
4 MB, using two 1 Mbit SIMMs on the memory expansion card and setting the jumper to "SIMM INSTALLED"

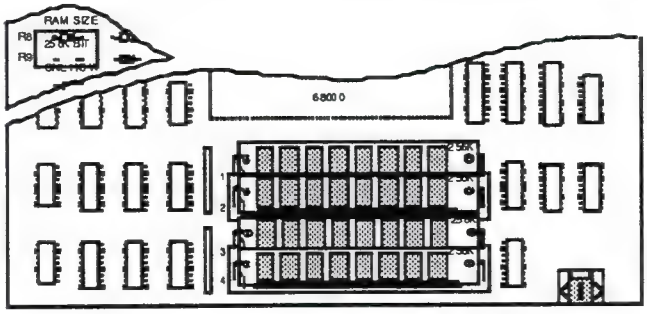
When adding SIMMs to the memory expansion card, use either two 256 Kbit or two 1 Mbit parts rated at 120 ns or faster.



System Memory Size: 512K

SIMMs Configuration
Row 1 (SIMMs 1 & 2): 256K
Row 2 (SIMMs 3 & 4): Not Installed

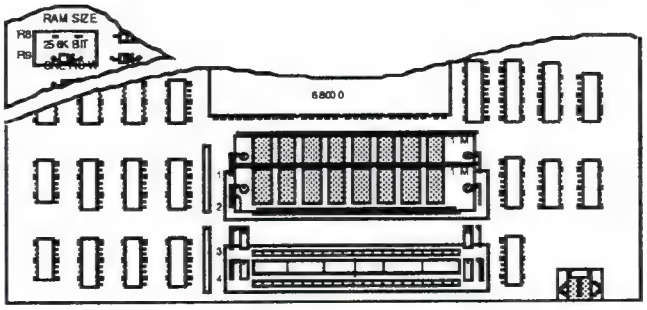
RAM SIZE Resistors
256 Kbit (R8): 150 Ohms
One Row (R9): 150 Ohms



System Memory Size: 1 MB

SIMMs Configuration
Row 1 (SIMMs 1 & 2): 256K
Row 2 (SIMMs 3 & 4): 256K

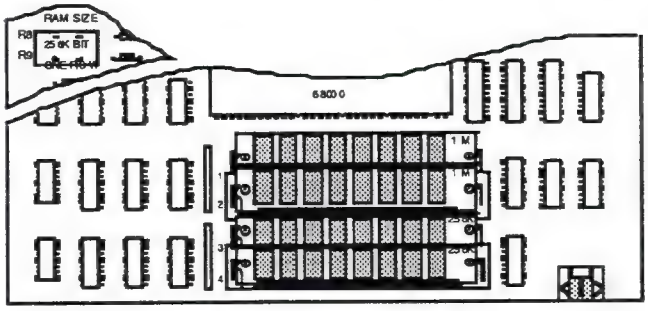
RAM SIZE Resistors
256 Kbit (R8): 150 Ohms
One Row (R9): Not Installed



System Memory Size: 2 MB

SIMMs Configuration
Row 1 (SIMMs 1 & 2): 1 MB
Row 2 (SIMMs 3 & 4): Not Installed

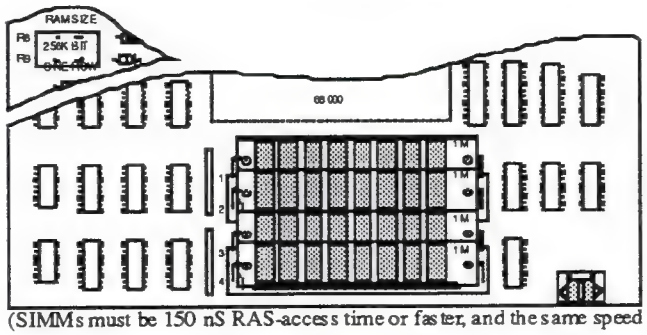
RAM SIZE Resistors
256 Kbit (R8): Not Installed
One Row (R9): 150 Ohms



System Memory Size: 2.5 MB

SIMMs Configuration
Row 1 (SIMMs 1 & 2): 1 MB
Row 2 (SIMMs 3 & 4): 256K

RAM SIZE Resistors
256 Kbit (R8): Not Installed
One Row (R9): Not Installed



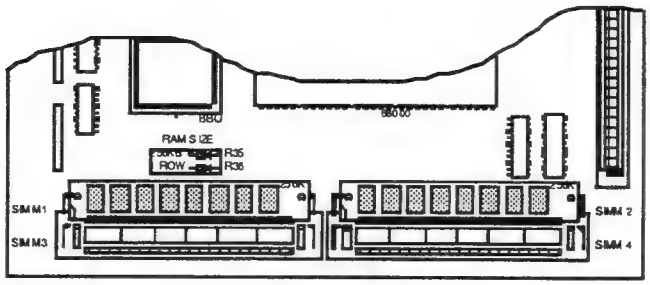
System Memory Size: 4 MB

SIMMs Configuration
Row 1 (SIMMs 1 & 2): 1 MB
Row 2 (SIMMs 3 & 4): 1 MB

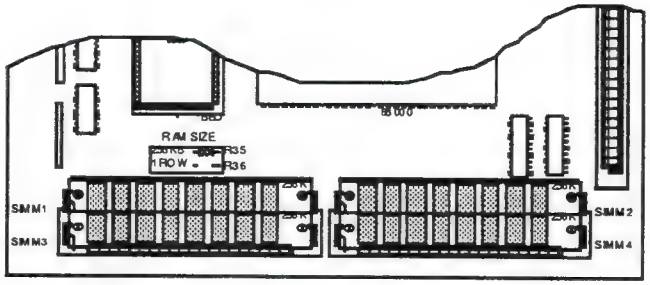
RAM SIZE Resistors
256 Kbit (R8): Not Installed
One Row (R9): Not Installed

(SIMMs must be 150 nS RAS-access time or faster, and the same speed within a row.)

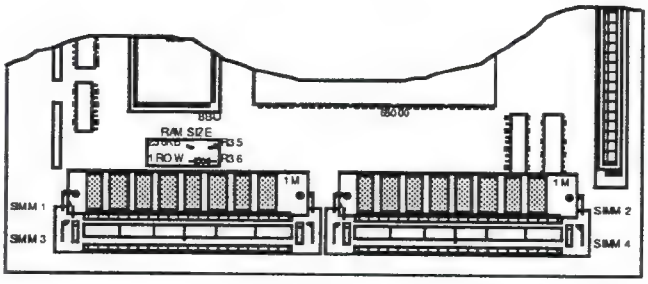
Figure 1–Macintosh Plus Memory Configurations



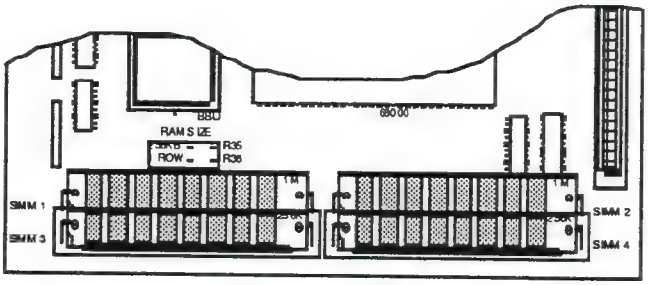
System Memory Size	512K
SIMMs Configuration	
Row 1 (SIMMs 1 & 2)	256K
Row 2 (SIMMs 3 & 4)	Not Installed
RAM SIZE Resistors	
256 Kbit (R35)	150 Ohms
One Row (R36)	150 Ohms



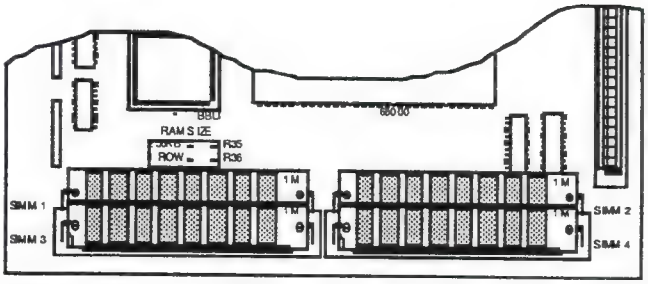
System Memory Size	1 MB
SIMMs Configuration	
Row 1 (SIMMs 1 & 2)	256K
Row 2 (SIMMs 3 & 4)	256K
RAM SIZE Resistors	
256 Kbit (R35)	150 Ohms
One Row (R36)	Not Installed



System Memory Size	2 MB
SIMMs Configuration	
Row 1 (SIMMs 1 & 2)	1 MB
Row 2 (SIMMs 3 & 4)	Not Installed
RAM SIZE Resistors	
256 Kbit (R35)	Not Installed
One Row (R36)	150 Ohms



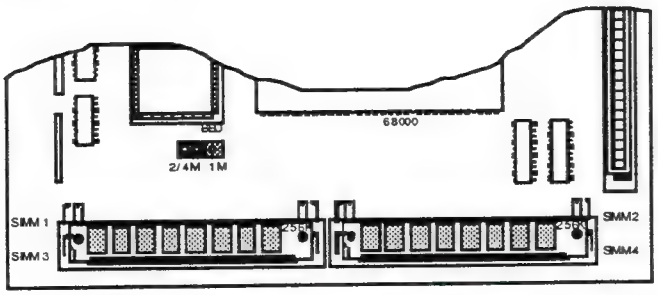
System Memory Size	2.5 MB
SIMMs Configuration	
Row 1 (SIMMs 1 & 2)	1 MB
Row 2 (SIMMs 3 & 4)	256K
RAM SIZE Resistors	
256 Kbit (R35)	Not Installed
One Row (R36)	Not Installed



System Memory Size	4 MB
SIMMs Configuration	
Row 1 (SIMMs 1 & 2)	1 MB
Row 2 (SIMMs 3 & 4)	1 MB
RAM SIZE Resistors	
256 Kbit (R35)	Not Installed
One Row (R36)	Not Installed

(SIMMs must be 150 nS RAS-access time or faster, and the same speed within a row.)

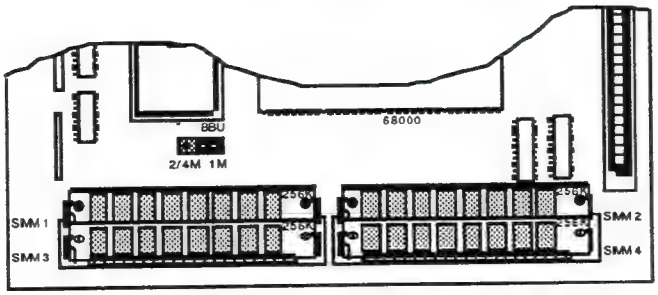
Figure 2–Macintosh SE Memory Configurations



System Memory Size 512K

SIMMs Configuration
Row 1 (SIMMs 1 & 2): Not Installed
Row 2 (SIMMs 3 & 4): 256 K

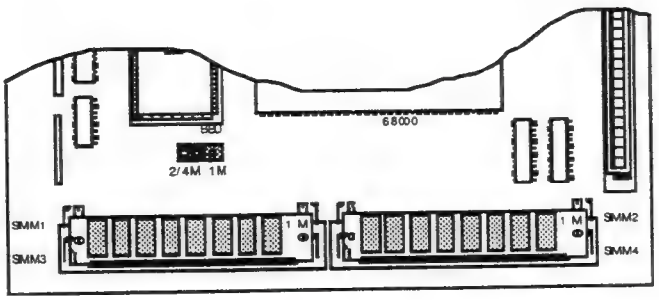
Jumper on 2/4M



System Memory Size 1 MB

SIMMs Configuration
Row 1 (SIMMs 1 & 2): 256 K
Row 2 (SIMMs 3 & 4): 256 K

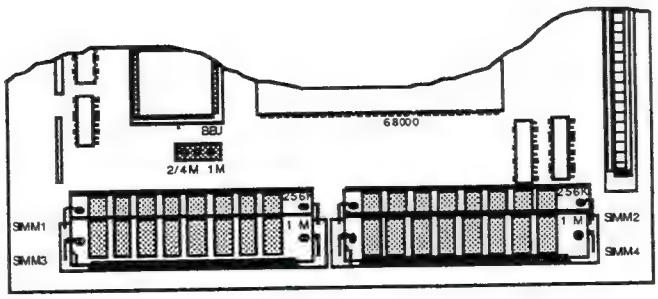
Jumper on 1M



System Memory Size 2 MB

SIMMs Configuration
Row 1 (SIMMs 1 & 2): Not Installed
Row 2 (SIMMs 3 & 4): 1 MB

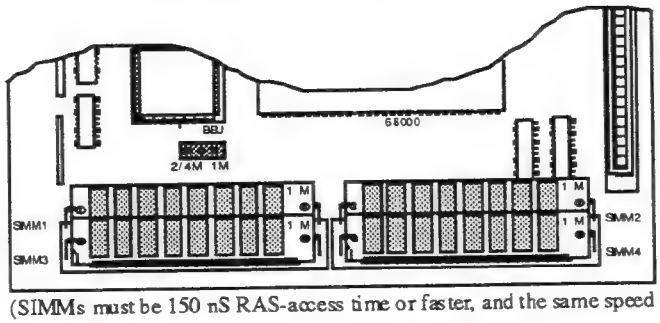
Jumper on 2/4M



System Memory Size 2.5 MB

SIMMs Configuration
Row 1 (SIMMs 1 & 2): 256K
Row 2 (SIMMs 3 & 4): 1 MB

Jumper off



System Memory Size 4 MB

SIMMs Configuration
Row 1 (SIMMs 1 & 2): 1 MB
Row 2 (SIMMs 3 & 4): 1 MB

Jumper off

(SIMMs must be 150 nS RAS-access time or faster, and the same speed within a row.)

Figure 3–Macintosh SE (with jumper) Memory Configurations

Macintosh SE/30, II, IIx, and IIcx

Since these machines use a 32-bit data bus with eight-bit SIMMs, you must always upgrade memory in four SIMM chunks. The eight SIMM connectors are divided into two banks of four SIMM slots, Bank A and Bank B.

On the Macintosh SE/30, Bank A is located next to the ROM SIMM while Bank B is next to the 68882 coprocessor. On the Macintosh II and IIx, Bank A is the bank closest to the edge of the board, while on the Macintosh IIcx, Bank A is the bank closest to the disk drives and power supply. Refer to Figure 5 for the proper locations of Banks A and B on the SE/30, II, and IIx, and refer to Figure 6 for the proper locations on the IIcx.

Unlike the Macintosh Plus and the Macintosh SE, the Macintosh II and IIx have no resistors to cut and no jumpers to set; you need only install the SIMMS in the correct banks and you'll be up and running. You can implement the following configurations:

1 MB, using four 256 Kbit SIMMs in Bank A
2 MB, using eight 256 Kbit SIMMs in Banks A and B
4 MB, using four 1 Mbit SIMMs in Bank A
5 MB, using four 1 Mbit SIMMs in Bank A and four 256 Kbit SIMMs in Bank B
8 MB, using eight 1 Mbit SIMMs in Banks A and B
>8 MB: see the 32-bit addressing information below

Again, it is important to make sure the right size SIMMs are in the right Bank; when you are using a combination of SIMMs, the larger SIMMs (in terms of Mbits) must typically be in Bank A (see the exception below). When you are using only four SIMMs, they must be in Bank A as well.

32-Bit Addressing With the Macintosh SE/30, II, IIx, and IIcx

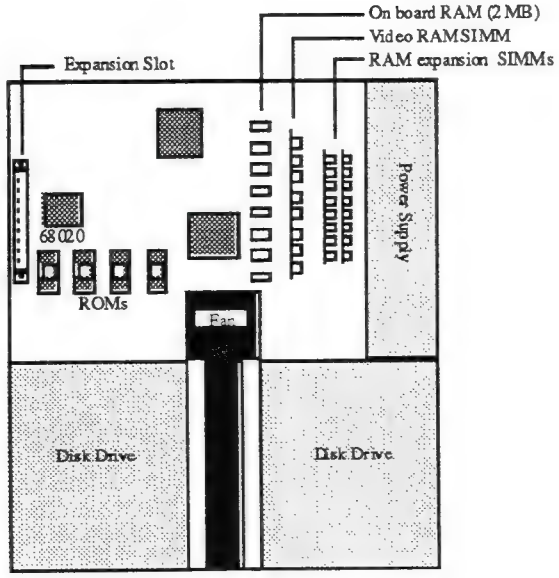
The Macintosh SE/30, II, IIx, and IIcx ROMs are not capable of 32-bit addressing. These models can overcome this limitation, however, by using the appropriate system software. A/UX is a 32-bit operating system, as is System 7 when used in conjunction with MODE32 or when used on a Macintosh with 32-bit clean ROMs.

To have more than 8 MB of RAM in a Macintosh II or IIx, special 120 ns PAL SIMMs are required. These SIMMs incorporate PAL logic chips that overcome problems caused by the refresh logic on the Macintosh II and IIx. In addition, a PMMU is required on the Macintosh II. Please refer to the end of this Note ("4 MBit DRAMs in Revolt") for more information on this subject.

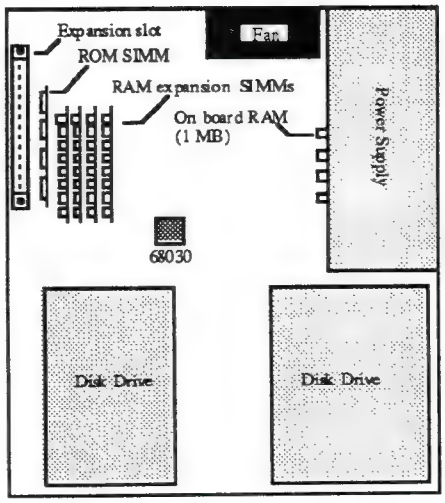
Due to an undocumented feature in the ROM firmware shipped with the original Macintosh II, a Macintosh II with original ROMs is limited to using SIMMs no larger than 1 MB in Bank A. Large SIMMs can only be put in Bank B (that is, 4 and 16 MB SIMMs). Remember that if Bank B is to be used at all, Bank A must be populated first. As a result of this limitation, the largest memory configuration on an unmodified Macintosh II using 1 MB SIMMs in Bank A and 4 MB SIMMs in Bank B is 20 MB. This problem is avoided if you've installed the SuperDrive upgrade kit, which includes a set of Macintosh IIx ROMs. The Macintosh IIx ROMs can handle 4 MB SIMMs, and expect the presence of a SWIM chip in place of the old IWM.

The theoretical maximum memory that a Macintosh SE/30, IIx, IIcx (and II with IIx ROMs) can address is 128 MB using 16 MB SIMMs.

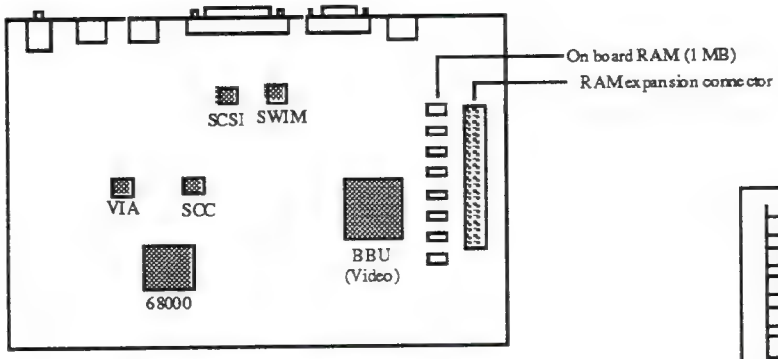
Please remember that the use of large SIMM sizes with the Macintosh hardware line has not yet been tested thoroughly. It is mentioned here for your consideration and should be considered theoretical until we have been able to further test all of these possible configurations.



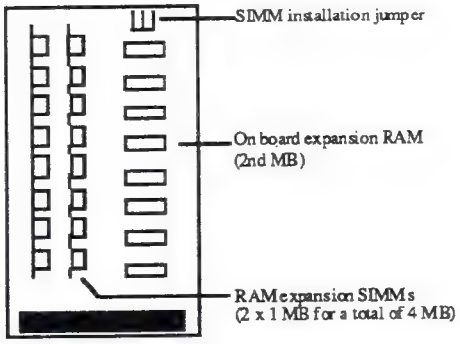
Macintosh LC
(RAM SIMMs must be 100 nS RAS-access time or faster.)



Macintosh IIsi
(SIMMs must be 100 nS RAS-access time or faster.)

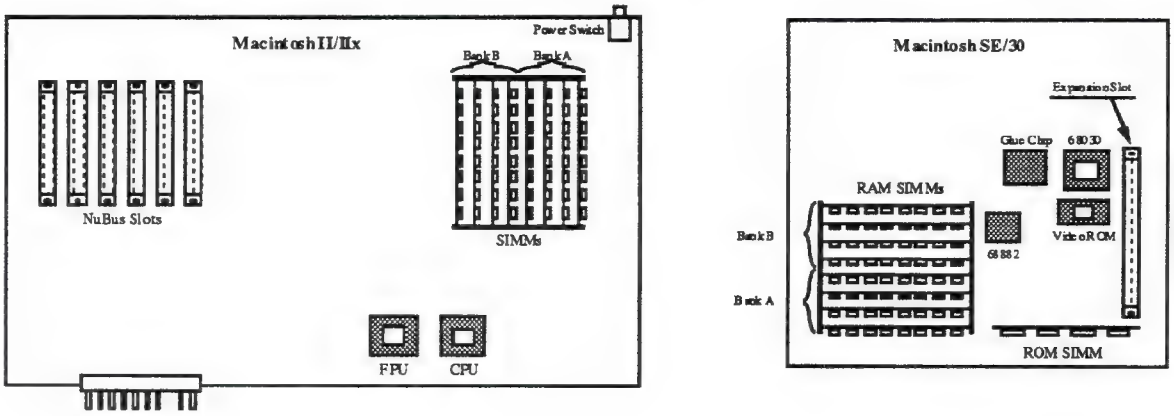


Macintosh Classic
(SIMMs must be 120 nS RAS-access time or faster.)



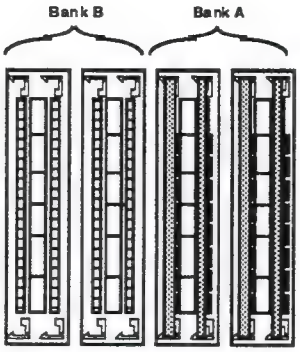
Macintosh Classic RAM expansion board

Figure 4–Macintosh Classic, LC, and IIsi

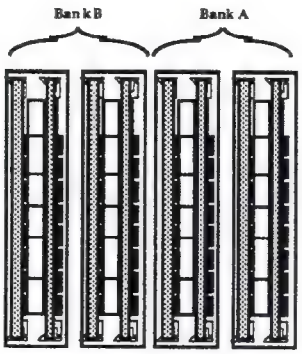


(SIMMs must be 120 nS RAS-access time or faster, and the same speed within a row.).

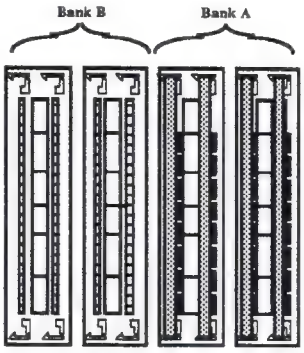
Macintosh II, IIx, and Macintosh SE/30 memory configurations are identical.



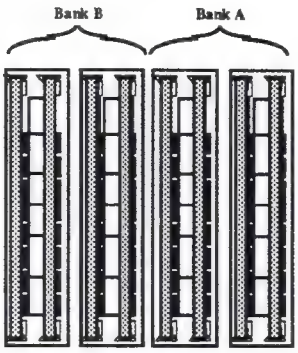
System Memory Size: 1 MB
Bank A: 4 x 256K SIMMs
Bank B: Empty



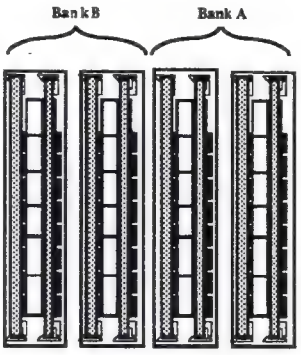
System Memory Size: 2 MB
Bank A: 4 x 256K SIMMs
Bank B: 4 x 256K SIMMs



System Memory Size: 4 MB
Bank A: 4 x 1 MB SIMMs
Bank B: Empty



System Memory Size: 5 MB
Bank A: 4 x 1 MB SIMMs
Bank B: 4 x 256K SIMMs



System Memory Size: 8 MB
Bank A: 4 x 1 MB SIMMs
Bank B: 4 x 1 MB SIMMs

Figure 5–Macintosh SE/30, II, and IIx Memory Configurations

Macintosh LC

The Macintosh LC uses a 16-bit data bus with 8-bit SIMMs, so upgrades must always be performed two SIMMs at a time. The LC has two SIMM connectors that are used as a single additional RAM bank (see Figure 4) in addition to the 2 MB already soldered to the motherboard. The following memory configurations can be implemented by installing SIMM pairs in this additional bank:

2 MB, using four 1 Mbit x 4 DRAMs soldered to the motherboard
4 MB, using two 1 Mbit SIMMs in the SIMM connectors
6 MB, using two 2 Mbit SIMMs in the SIMM connectors
10 MB, using two 4 Mbit SIMMs in the SIMM connectors

The Macintosh LC requires 100 ns or faster SIMMs.

Macintosh IIsi

The Macintosh IIsi is similar to the SE/30, II, IIx, and IIcx in that it uses a 32-bit data bus with 8-bit SIMMs; you must always upgrade memory in four SIMM chunks. The IIsi differs in that it only has one SIMM bank instead of two (see Figure 4).

If future 16 Mbit DRAMs are compatible with the current refresh frequency, then the IIsi will support 16 Mbit SIMMs, enabling a RAM configuration of 65 MB (4 x 16 MB + 1 MB). The IIsi requires 100 ns or faster SIMMs.

Macintosh IIci

The Macintosh IIci motherboard layout is somewhat different from the IIcx, but the location of the RAM SIMMs is unchanged. Bank A is still the bank closest to the disk drives. Refer to Figure 6 for the proper locations of Banks A and B on the IIci.

The IIci has a much improved RAM interface and allows a great deal more freedom when installing SIMMs. Banks A and B are interchangeable, meaning that when mixing two sizes of RAM, the larger SIMMs do not necessarily have to go in Bank A. In fact, for best performance when using on-board video, Apple recommends that the smaller SIMMs be installed in Bank A. Note, however, that if on-board video is used, then RAM **must** be present in Bank A.

The IIci **requires** that SIMMs be 80 ns
time or faster and the same speed within a row. You can implement the following memory configurations with 256K and 1 MB SIMMs:

1 MB using four 256 Kbit SIMMs in Bank A or in Bank B
2 MB using eight 256 Kbit SIMMs in Banks A and B
4 MB using four 1 Mbit SIMMs in Bank A or in Bank B
5 MB using four 256 Kbit SIMMs in Bank A and four 1 Mbit SIMMs in Bank B
5 MB using four 1 MBit SIMMs in Bank A and four 256 Kbit SIMMs in Bank A
8 MB using eight 1 Mbit SIMMs in Banks A and B

The 1 MB and 4 MB configurations using only Bank B are not compatible with on-board video, since Bank A must contain memory when using on-board video. The first 5 MB configuration (with 256 Kbit SIMMs in Bank A) is recommended for 5 MB configurations using on-board video.

Parity RAM

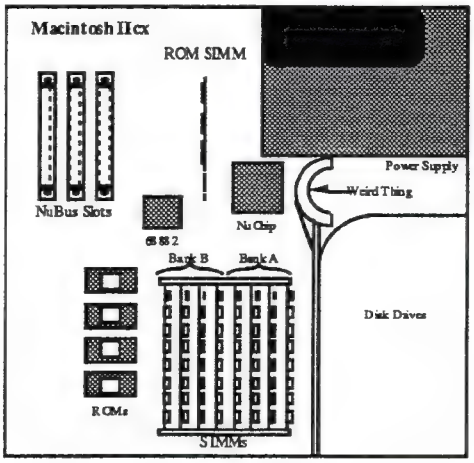
Some specially ordered versions of the Macintosh IIci are equipped with a PGC chip and support parity for RAM error detection. These machines require parity RAM. SIMMs for these machines are nine bits wide instead of eight, so there is generally an extra RAM IC on the SIMM. There is no difference in the installation of 256K x 9 or 1M x 9 SIMMs.

Macintosh Portable

Memory expansion on the Macintosh Portable is different from other members of the Macintosh family since the Portable uses memory expansion cards in place of SIMMs. The base Portable is equipped with 1 MB of RAM on the motherboard and has one RAM expansion card slot. Apple currently supplies a 1 MB memory expansion kit that takes the Portable to 2 MB total. Apple and third-party developers may produce higher-capacity expansion boards (2 MB to 8 MB) in the future.

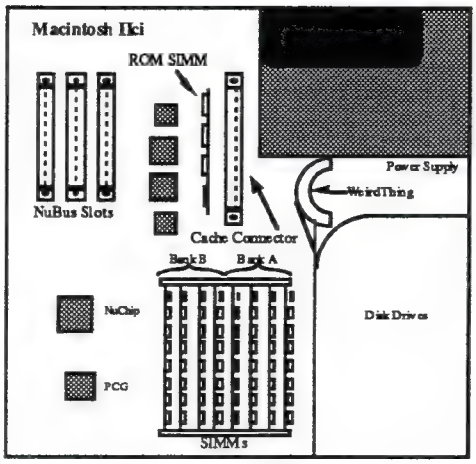
Since the Portable has only one RAM expansion slot, you may use only one memory expansion board at a time. This limit means that a 1 MB expansion board would have to be completely replaced by a higher-capacity board when it became available.

Total RAM for the Portable will always be 1 MB plus the size of your one RAM expansion board (if installed). Refer to Figure 6 for the location of the RAM expansion slot.



(SIMMs must be 120 nS RAS-access time or faster, and the same speed within a row.)

Macintosh IIcx memory configurations are identical to the II, IIx, and SE/30.



(SIMMs must be 80 nS RAS-access time or faster, and the same speed within a row.)

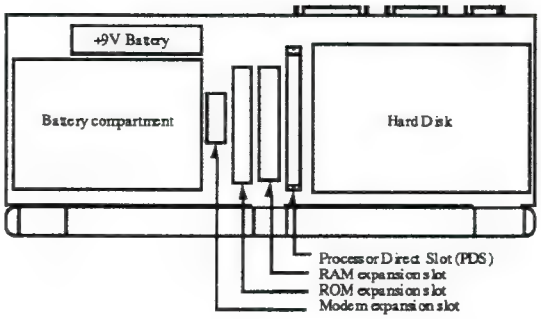


Figure 6–Macintosh IIcx, IIci, and Portable Memory Configurations

Macintosh IIfx

The Macintosh IIfx motherboard layout has its SIMMs located in the same general area as the IIx, but they are oriented transversely. Bank A is the bank closest to the rear of the machine; bank B is closest to the main processor. Refer to Figure 7 for the proper memory bank locations.

The IIfx has a RAM SIMM interface similar to that of the IIcx, et al.: when you are using a combination of SIMMs, the larger SIMMs (in terms of Mbits) must be in Bank A. When you are using only four SIMMs, they must be in Bank A as well. The description in the *Guide to the Macintosh Family Hardware*, Second Edition inaccurately states the larger SIMMs can be placed in either bank.

The IIfx **requires** that SIMMs be 80 ns RAS-access time or faster and the same speed within a row. You can implement the following memory configurations with 1 and 4 MB SIMMs (256K address-depth SIMMs are not supported):

4 MB using four 1 Mbit SIMMs in Bank A
8 MB using eight 1 Mbit SIMMs in Banks A and B
16 MB using four 4 Mbit SIMMs in Bank A
20 MB using four 4 Mbit SIMMs in Banks A and four 1 Mbit SIMMs in Bank B
32 MB using eight 4 Mbit SIMMs in Banks A and B

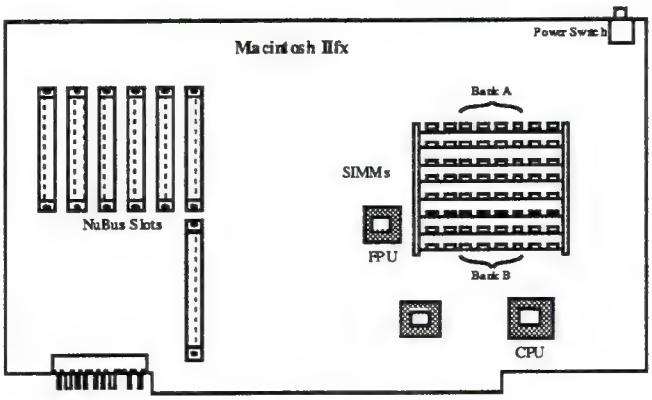
Parity RAM

Parity RAM requirements are as follows: if using 1 MB or 4 MB SIMMs, the RAM speed must be 60 ns. However, the parity circuit programmable array that goes on the motherboard as well as the parity PALs that go on the SIMMs are proprietary to Apple—their equations are not expected to be released to developers. Because of this proprietary design, Apple does not recommend third-party development of parity products.

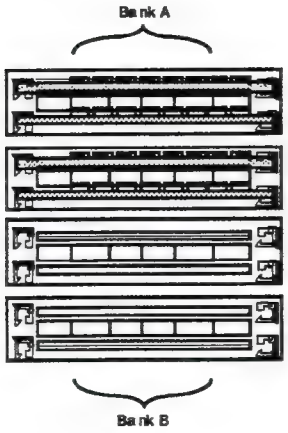
RAM SIMM Drawings

The IIfx has 64-pin SIMMs, which are different from previous Macintosh models. Developers can request mechanical drawings and electrical specifications of the IIfx RAM SIMM modules from DTS. Please send the request with a mailing address and include the words "IIfx SIMM information request" in the title of the electronic mail request or letter to facilitate handling.

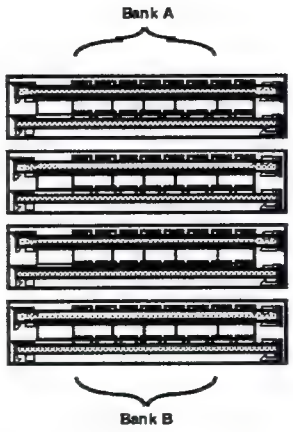
Warning: To avoid degradation of signal quality, it is critical to adhere to the strict timing parameters of the IIfx and to use a good layout that takes high-speed circuits into account.



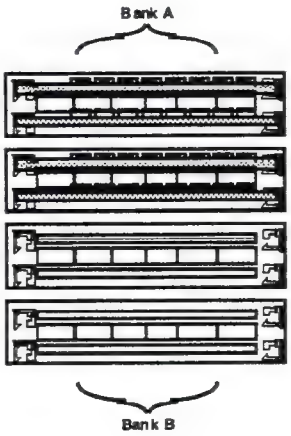
(SIMMs must be 80 nS RAS-access time or faster, and the same speed within a row).



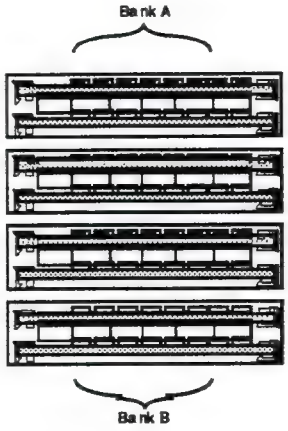
System Memory Size: 4 MB
Bank A: 4 x 1 MB SIMMs
Bank B: Empty



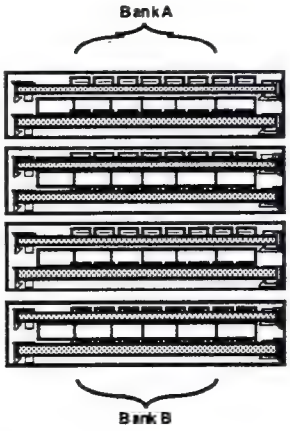
System Memory Size: 8 MB
Bank A: 4 x 1 MB SIMMs
Bank B: 4 x 1 MB SIMMs



System Memory Size: 16 MB
Bank A: 4 x 1 MB SIMMs
Bank B: Empty



System Memory Size: 20 MB
Bank A: 4 x 4 MB SIMMs
Bank B: 4 x 1 MB SIMMs



System Memory Size: 32 MB
Bank A: 4 x 4 MB SIMMs
Bank B: 4 x 4 MB SIMMs

Figure 7–Macintosh IIfx Memory Configurations

Macintosh Quadrs 900

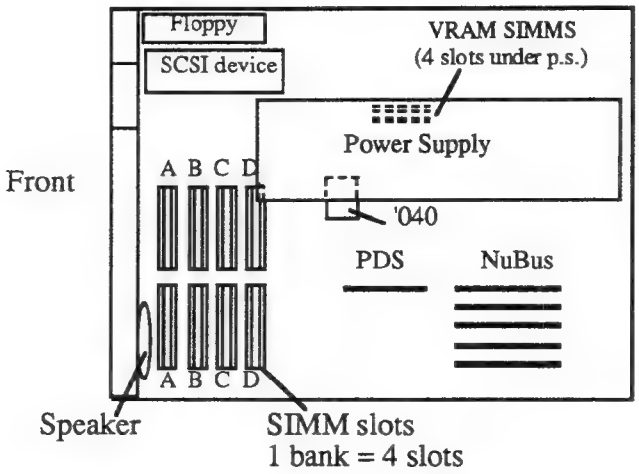


Figure 8 - View of the Macintosh Quadra 900 With Case Open

The memory control unit (MCU) controls four banks of dynamic RAM, for a total of 16 SIMM slots. Each bank accepts standard 80 ns SIMMs containing 1 MB, 4 MB, and perhaps 16 MB SIMMs (256K and 2 MB are not supported), giving total memory sizes from 4 MB to 16 MB for each bank (64 MB if 16 MB SIMMs work). Therefore, the Macintosh Quadra 900 could have a total of 64 MB when using currently available 4 MB SIMMs. 16 MB SIMMs have not been thoroughly tested on the Quadra 900 and therefore cannot be listed as a possible configuration. The Macintosh Quadra 900 can also use 60 ns SIMMs, but the MCU is programmed for 80 ns DRAM, so a 60 ns SIMM wouldn't improve the speed.

If one slot in a given bank is filled, then all slots in that particular bank must be populated with the same size SIMM. It is not possible to mix the speed of RAM, even between banks. The order that the banks are populated does not matter (for example, it is acceptable to have four 1 MB SIMMs in Bank B, and four 4 MB SIMMs in Bank D).

Note: When large amounts of DRAM are installed, the memory check upon startup is lengthy and can cause users to think that the machine isn't functioning. There is no software indication that the machine is running memory checks.

Macintosh Quadra 700

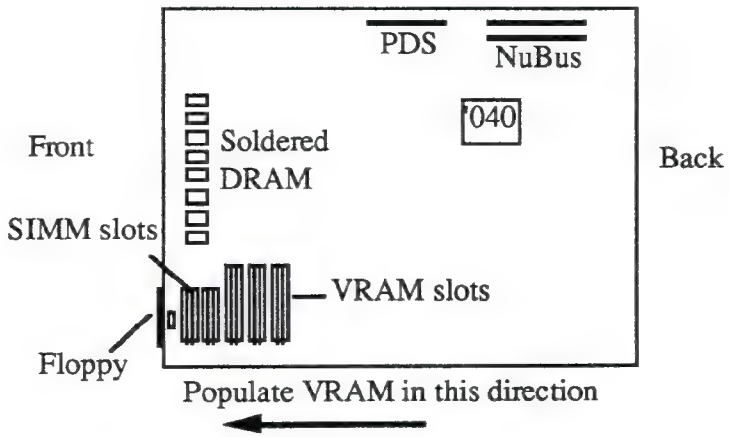


Figure 9 - View of the Macintosh Quadra 700 With Case Open

The memory control unit (MCU) IC controls two banks of dynamic RAM on the Macintosh Quadra 700. The first bank is soldered down at the factory, and fixed at 4 MB. The additional bank accepts standard 80 ns SIMMs containing 1 MB, 4 MB, and perhaps 16 MB (256K and 2 MB are not supported), giving total memory sizes from 4 MB to 16 MB for each bank. Therefore the Macintosh Quadra 700 could have a total of 20 MB when using SIMMs that are currently available. 16 MB SIMMs have not been tested on the Quadra 700 and therefore cannot be listed as a possible configuration. The Macintosh Quadra 700 can also use 60 ns SIMMs, but the MCU is programmed for 80 ns DRAM, so a 60 ns SIMM wouldn't improve the speed. If one slot in a given bank is filled, then all slots in the bank must be populated. It is not possible to mix the speed of RAM SIMMs on the Quadra 700.

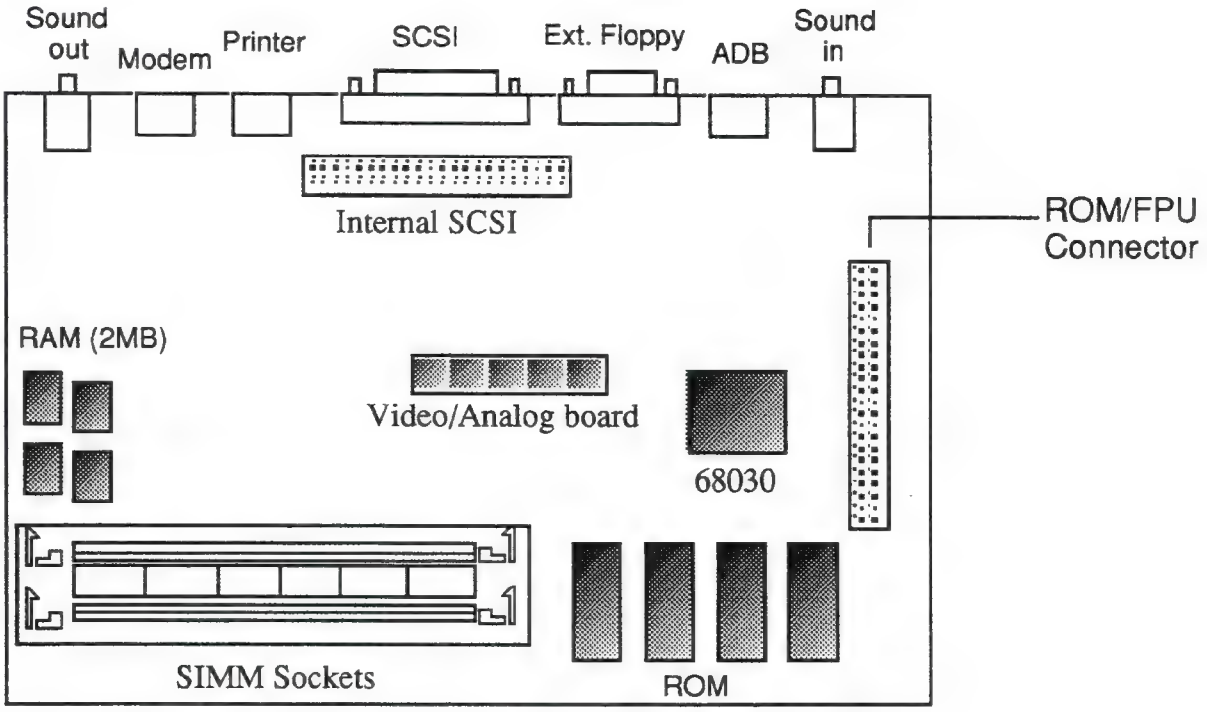
Note: Due to the location of the SIMM slots on the Macintosh Quadra 700, it is unlikely that third-party vendors will be able to develop 16 MB SIMMs that work on this machine. The placement of the SIMM slots (under the hard drive) is unfortunate, but necessary due to the logic board real estate.

Macintosh Classic II

The Macintosh Classic II can support up to 10 MB of system RAM. The logic board includes 2 MB soldered and 2 SIMM sockets that can accommodate 1, 2, or 4 MB SIMMS for possible system configurations of 2, 4, 6, or 10 MB. The Classic II requires 100 ns or faster access time for SIMMs to be compatible.

Possible memory configurations:

Built-in	+	SIMM size	x2	=	Total RAM (MB)
2		0			2
2		1			4
2		2			6
2		4			10



Macintosh Classic II
(SIMMs must be 100 ns RAS-access time or faster.)

Macintosh PowerBook 140 and Macintosh PowerBook 170

The Macintosh PowerBook 140 and 170 ship with 2 MB of PSRAM (pseudo-static) on the daughter board. The RAM is arranged physically as four 4 Mbit chips of 512K x 8-bits each. Additionally, an expansion slot allows RAM to be expanded to a total of 8 MB. Both notebook systems use 100 ns PSRAM. The Macintosh PowerBook 140 has one wait state (four clock cycles). The Macintosh PowerBook 170 has two wait states (five clock cycles). PSRAM needs to be refreshed, and the refresh is accomplished by circuitry in the CPU Glue Logic ASIC. The refresh requirement causes a 2% reduction in performance over SRAM. However, PSRAM uses less current in sleep mode and costs less than SRAM.

The PowerBook computers contain a 70-pin RAM expansion connector (slot) that supports RAM expansion card sizes of 2, 4, and 6 MB. Apple offers a 2 MB and 4 MB memory card.

Note: If a RAM expansion card is designed correctly, it will work in all of the PowerBook computers. The 68030-based PowerBooks have a 32-bit data bus whereas the 68HC000-based machine (the PowerBook 100) has only a 16-bit data bus. If the expansion card is designed as a 32-bit device it will only work in the PowerBook 140 and 170, but if the data lines and chip select lines are in the correct location on the card, users can use the same card in either machine without loss of performance. The separated chip select lines are necessary for the 68HC000-based machine because it can get access to only 16 bits at a time. The Macintosh PowerBook 140 and 170 do not require separated chip select lines because both have a 32-bit data bus; therefore the lines are tied back together on the computer's main logic board.

The Apple PowerBook RAM cards will work in all three PowerBook systems. If purchasing RAM from third-party vendors, make sure it will work in the 68HC000–based portable as well.

RAM is always contiguous because only one size of RAM chip (4 Mbits) is used. As a result, the amount of memory the computer has doesn't have to be determined by the software. The RAM array is nominally located in the system memory map between addresses $0000 0000 and $0020 0000 (up to $0080 0000 in an 8 MB system), except following a system reset or sleep cycle, at which time it is overlaid by system ROM. However, the overlay is removed following access to normal ROM space, and the RAM space is then accessible. Both RAM and ROM memory spaces provide DSACK signals to the processor even if memory is not actually installed.

RAM wait states: access to the RAM from the main processor requires 100 ns PSRAM and two processor wait states (five clock cycles per RAM access). The CPU Glue Logic custom chip includes special circuitry that performs the refresh function.

Battery backup: both main and expansion RAM memories are backed up when the computer is in the sleep mode. This means that when the computer is not in use, the contents of the memory array are retained as long as the battery remains charged.

Note: When the battery is removed RAM contents are lost.

Unlike the Macintosh PowerBook 100 and Macintosh Portable, when the battery is removed, the contents of RAM are lost. The user must shut down the unit before replacing the battery. The backup battery in the Macintosh PowerBook 140 and 170 computers supply power only to the RTC chip (the clock).

Macintosh PowerBook 100

The system comes with 2 MB of PSRAM (pseudo-static RAM) . The RAM is arranged as four 4 Mbit chips of 512 by 8 bits each. The memory chips have an access and cycle time of 100 ns. There are no processor wait states to RAM unless the requested location in the pseudo-static RAM is being refreshed.

The system RAM can be expanded via a new 70-pin RAM expansion connector. The expansion slot can be filled with card sizes of 2 MB, 4 MB, or 6 MB. The system will automatically determine the card's memory size. These memory expansion cards can be used with the Macintosh PowerBook 140 and 170 computers.

System RAM is always powered; therefore RAM disks will be saved even after shutdown (similar to the Macintosh Portable). RAM will be maintained by three lithium batteries during a main battery exchange.

4 Mbit DRAMs in Revolt

When the Macintosh II was originally designed, Apple engineers intended for it to accept large amounts of memory in the form of 4 MB and 16 MB DRAM SIMMs. That was in 1986, when 1 Mbit DRAM was difficult to find and the higher-density chips did not yet exist. The engineers anticipated the pinouts of the yet to be introduced 4 MB SIMMs and provided all the necessary hardware and address multiplexing to allow installation of these parts when they became available.

Woe that Cupertino is not Camelot, James Brown is still on probation, and 4 MB SIMMs do not work as advertised in most cases. This is the story of the Revolt of the 4 MB DRAM SIMMs.

Preliminary Notes

Before diving into the problem with 4 Mbit DRAMs, there is some preliminary ground that must be covered.

First, there are a couple ways to construct a 4 MB SIMM. Using old technology, it is possible to cram together 32 DRAM ICs of 1M x 1 density. Using new technology, it only takes eight 4M x 1 ICs, resulting in a much smaller, lower-power module. If a 4 MB SIMM is of the large, so-called composite type (that is, it is constructed of 32 1 Mbit ICs), then everything is fine except on the original Macintosh II. Please refer to page 7 of this Tech Note for more information on Macintosh II RAM.

With the FDHD SuperDrive upgrade kit installed, the Macintosh II is on equal footing with the Macintosh IIx. That is, SIMMs made exclusively of the new 4 Mbit ICs still won't work, regardless of whether you are using a Macintosh II or IIx; therefore, for the remainder of this discussion, *Macintosh II* is used to refer to not only the original Macintosh II, but also the IIx.

Subsequent Macintosh models have revised ROMs that recognize 4 MB SIMMs.

The 4 Mbit Problem

DRAM ICs are now available in 4 Mbit density, but they come with a very nasty surprise. JEDEC, the committee overseeing the standardization of new solid-state devices, has added an additional built-in test mode to high-density DRAMs. The test mode is invoked by a sequence of electrical signals that was ignored by earlier-generation DRAM. The crux of the situation is this: under certain conditions, the Macintosh II unwittingly activates this new test mode and large amounts of memory become very forgetful.

More Specifically . . .

Those who are interested in the specific phenomenon occurring within the memory ICs should consult the detailed technical data supplied by the DRAM manufacturers. This Note only explains how the Macintosh II offends this new feature of the 4 Mbit DRAM, and hence, what might be done to work around the problem.

The Macintosh II uses /CAS-before-/RAS refresh cycles to keep RAM up to date on its contents. For 1 Mbit DRAM, the state of the /W control line is ignored during this type of refresh cycle. No longer. DRAM of the 4 Mbit variety goes off into test mode if /W is asserted (low, so that the RAM thinks it is write-enabled) during a /CAS-before-/RAS refresh cycle. The problem with the Macintosh II is that /W is the same signal as the MPU R/W line, and if the MPU is writing to an I/O address or a NuBus™ card concurrently with a refresh cycle, all the conditions are right for a waltz into test mode. Unfortunately, this condition is not all that unusual, since video card accesses qualify.

Consolation for SIMM manufacturers: SIMMs constructed with an on-board PAL are not necessarily Macintosh II-specific. SIMMs constructed in this manner should work without modification in any usage calling for 4 MB SIMMs (except in the unlikely event that the new test mode is required).

The Salvage Process

All is not necessarily lost, and although the situation is ugly, there is still a way to use 4 Mbit DRAM ICs to construct 4 MB SIMMs that work in the Macintosh II. A solution lies in the addition of a ninth IC to the SIMM. Programmed with suitable logic, a high-speed (-D or -E suffix) PAL on the SIMM itself can recognize and intercept /CAS-before-/RAS refresh cycles and set /W appropriately before any damage is done. More or less, the PAL becomes an intelligent buffer between the MPU read/write line and the DRAM write-enable lines. When the PAL senses a refresh cycle commencing, it holds /W high, ensuring that the ICs are not corrupted by the potentially dangerous processor-generated R/W signal.

What's the Point?

You have overcome all the problems discussed in this section and have working 4 Mbit SIMMs installed in your Macintosh. You probably have at least 20 MB of RAM. What can you do with all of it? Create lots of huge 32-bit PICTs and edit them all simultaneously? Model and animate Bay Area weather patterns in Mathematica™? Yes! But, you have to use the appropriate system software to address this memory. Also, if you're running in 32-bit addressing mode, the applications that you desire to use need to be 32-bit clean. For more information on 32-bit cleanliness and addressing, please see Technical Notes #212 and #213.

Under System 7.0, applications can finally access additional physical memory over and above 8 MB. As mentioned previously in this Tech Note, the 32-bit addressing mode of System 7 requires either a Macintosh with 32-bit clean ROMs (listing is on page 2), or else the 32-bit software solution provided by the MODE32 system extension. A/UX is an alternative that can use up to 256K of RAM on Macintosh computers that support A/UX. Many manufacturers of large SIMMs also offer RAM disks. This is a volatile form of storage, but can certainly be useful for I/O intensive operations.

Other Permutations

The problem with 4 Mbit DRAM is not limited to 4 MB SIMMs. It is the 4 Mbit density of the individual RAM ICs that causes problems with certain machines. There exist 1 MB SIMMs constructed of only two 1M x 4 (4 Mbit) ICs. These do not work in a Macintosh II or IIx, any more than 4 MB SIMMs constructed of eight 4M x 1 ICs.

A few machines, namely the Macintosh Plus, Macintosh SE, and Macintosh Classic, depend on video accesses to refresh all of their DRAM. As the video circuitry accesses sequential locations through the video frame buffer, it simultaneously refreshes row after row of memory, eventually refreshing all 512 rows. Memory at the 4 Mbit density, however, is arranged as 1024 rows and there are not sufficient video accesses to refresh all 1024 rows. Chunks of memory simply go blank. Thus for a different reason, 4 Mbit DRAM parts are also not compatible with these older Macintosh hardware designs.

Executive Summary

Owners of the Macintosh Plus, SE, Classic, II, or IIx are all likely to have problems with any 1 MB SIMM carrying only two ICs, or any 4 MB SIMM carrying only 8 ICs. Any SIMM

constructed in one of these ways likely uses 4 Mbit density DRAM ICs and does not account for problems with the 4 Mbit test mode nor the video refresh strategy of older Macintosh designs.

Further Reference:

- *Inside Macintosh*, Volume V-1, Compatibility Guidelines
- *Guide to the Macintosh Family Hardware*, Second Edition
- Macintosh IIsi, LC, and Classic Developer Notes
- Macintosh Classic II, Macintosh PowerBook Family, and Macintosh Quadra Family Developer Notes
- Macintosh Technical Notes #212 and #213

NuBus™ is a trademark of Texas Instruments.
PAL is a trademark of Monolithic Memories, Inc.
Mathematica is a trademark of Wolfram Research, Inc.
MODE32™ is a trademark of Connectix Corporation.

#177: Problem with WaitNextEvent in MultiFinder 1.0

See also: Technical Note #158—
Frequently Asked MultiFinder Questions
Technical Note #180—MultiFinder Miscellanea

Written by: Jim Friedlander November 2, 1987
Updated: March 1, 1988

This Technical Note discusses a bug in `WaitNextEvent` in MultiFinder 1.0. This bug only occurs when `WaitNextEvent` is called from the background. This bug will be fixed in the next release of MultiFinder. **Change since 11/87:** the bug will be fixed in Systems with versions greater than `$04FF`.

In MultiFinder 1.0, applications that use `WaitNextEvent`:

```
FUNCTION WaitNextEvent(mask: INTEGER; VAR event: EventRecord;
                       sleep: LONGINT; mouseRgn: RgnHandle): BOOLEAN;

pascal Boolean WaitNextEvent(mask,event,sleep,mouseRgn)
    unsigned short mask;
    EventRecord *event;
    unsigned long sleep;
    RgnHandle mouseRgn;
```

should **not** call `WaitNextEvent` from the background with a value of `sleep` that is greater than `50`. This value has been determined empirically for a Macintosh II; larger values can be used on the Macintosh Plus and the Macintosh SE. If an application uses a large value of `sleep` when running in the background, MultiFinder 1.0 will hang under the following circumstances:

- The application that is calling `WaitNextEvent` with a large `sleep` value has been put in the background.
- That application becomes the foreground application when another application (including the desk accessory handler) quits.

If you use a value of `sleep` that is small enough, this problem will not occur. If you use an algorithm to calculate `sleep`, make sure that the maximum is clipped to `50`.

This problem will be fixed in the next release of MultiFinder. You can call `SysEnvirons` to test for the system version number. This bug will not happen when the System version is greater than `$04FF`.

#178: Modifying the Standard String Comparison

See also: The International Utilities

Written by: Mark Davis, Priscilla Oppenheimer — November 2, 1987

Updated: March 1, 1988

This technical note describes how to modify the standard string comparison by constructing an `itl2` resource. Developers may want to modify the standard string comparison if Apple's comparison doesn't meet their needs or if Apple has not written a string comparison routine for the language that concerns them.

General Structure

The `itl2` resource contains a number of procedures that are used for accurate comparison of text by the International Utilities Package. Refer to *Inside Macintosh*, volume V for an explanation of the algorithm used. The default `itl2` for standard English text, which does no special processing, has the following form:

```
     ; normal Include/Load statements
     Include 'hd:mpw:aincludes:ScriptEqu.a'
     Print        On,NoMDir

     String       AsIs

;-------------------------------------------------------------------------
;    dispatch table at the front of the code.
;-------------------------------------------------------------------------
Intl1             Proc
     With         IUSortFrame,IUStrData
HookDispatch
     dc.w         ReturnEQ-HookDispatch       ; InitProc = 0
     dc.w         ReturnEQ-HookDispatch       ; FetchHook = 2
     dc.w         ReturnEQ-HookDispatch       ; VernierHook = 4
     dc.w         ReturnEQ-HookDispatch       ; ProjectHook = 6
     dc.w         ReturnEQ-HookDispatch       ; ReservedHook1 = 8
     dc.w         ReturnEQ-HookDispatch       ; ReservedHook2 = 10
```

```
;----------------------------------------------------------------------
; Some common exit points
;----------------------------------------------------------------------
ReturnNE
     tst.w          MinusOne                    ; set cc NE
     rts
ReturnEQ
     cmp.w          d0,d0                       ; set cc EQ
     rts
;----------------------------------------------------------------------
     EndWith
     EndWith
     End
```

If modifications need to be made to the comparison process, then one or more of the dispatches will be modified to point to different routines:

```
     dc.w           InitProc-HookDispatch       ; InitProc = 0
     dc.w           FetchProc-HookDispatch      ; FetchHook = 2
     dc.w           VernierProc-HookDispatch    ; VernierHook = 4
     dc.w           ProjectProc-HookDispatch    ; ProjectHook = 6
```

There are a number of different changes that can be made to the comparison routines. Some of the common modifications include:

1. Comparing two bytes as one character
   Yugoslavian "l" < "lj" < "m"; Japanese... [InitProc, FetchProc]
2. Comparing characters in different order
   Norwegian "z" < "å" [ProjectProc]
3. Comparing one character as two
   German "ä" ≈ "ae" [ProjectProc]
4. Ignoring characters unless strings are otherwise equal:
   "blackbird" < "black-bird" < "blackbirds" [ProjectProc]
5. Changing the secondary ordering
   Bibliographic "a" < "A" [VernierProc]

The comparison hook procedures are all assembly language based, with arguments described below. Since the routines may be called once per character in both strings, the routines should be as fast as possible.

The condition codes are used to return information about the status of the hook routine. Typically the normal processing of characters will be skipped if the CCR is set to NE, so the default return should always have EQ set. Each of these routines has access to the stack frame (A6) used in the comparison routine, which has the following form:

```
IUSortFrame         Record      {oldA6},Decrement
result              ds.w        1
argTop              equ         *
aStrText            ds.l        1
bStrText            ds.l        1
aStrLen             ds.w        1
bStrLen             ds.w        1
argSize             equ         argTop-*
```

```
return          ds.l    1
oldA6           ds.l    1
aInfo           ds      IUStrData
bInfo           ds      IUStrData
wantMag         ds.b    1               ; 1-MagStrig 0-MagIdString.
weakEq          ds.b    1               ; Signals at most weak equality
msLock          ds.b    1               ; high byte of master ptr.
weakMag         ds.b    1               ; -1 weak, 1 strong compare
supStorage      ds.b    18              ; extra storage.
localSize       equ     *               ; frame size.
                EndR
```

There are three fields in this frame that are of interest for altering text comparison. The `supStorage` field is an area reserved for use by the comparison hook procedures as they see fit. The `aInfo` and `bInfo` records contain information about the current byte positions in the two compared strings A and B, and information about the status of current characters in those string. The `IUStrData` record has the following form:

```
IUStrData       Record  0
curChar         ds.w    1               ; current character.
mapChar         ds.w    1               ; projected character.
decChar         ds.w    1               ; decision char for weak equality
bufChar         ds.b    1               ; buffer for expansion.
justAfter       ds.b    1               ; boolean for AE vs ligature-AE.
ignChar         ds.b    1               ; flag: ignore char.
noFetch         ds.b    1               ; flag: no fetch of next.
strCnt          ds.w    1               ; length word.
strPtr          ds.l    1               ; current ptr to string.
                EndR
```

The Init Procedure

The Init Procedure is used to initialize the comparison process. The main use for this procedure is for double-byte scripts. As an optimization, the International Utilities will perform an initial check on the two strings, comparing for simple byte-to-byte equality. Thus any common initial substrings are checked before the Init procedure is called. The string pointers and lengths in the `IUStrData` records have been updated to point just past the common substrings.

Languages such as Japanese or Yugoslavian, which may consider two bytes to be one character, may have to back up one byte, as shown below.

```
;---------------------------------------------------------------------
; Routine       InitProc
; Input         A6          Local Frame
; Output        CCR         NE to skip entire sort (usually set EQ)
; Trashes       Standard regs:              A0/A1/D0-D2
; Function      Initialize any special international hooks.
;               Double-byte scripts must synchronize AInfo.StrPtr &
;                BInfo.StrPtr here!
;---------------------------------------------------------------------
; Note: this should also check for single-byte nigori or maru, as below
```

```
InitProc
     move.w        AStrLen(a6), d0            ; A length
     sub.w         AInfo.StrCnt(a6),d0        ; see if its changed
     beq.s         @FixB                      ; A is done if not
     sub.l         #2,sp                      ; return param
     move.l        AStrText(a6),-(sp)         ; textBuf
     move.w        d0,-(sp)                   ; textOffset
     _CharByte
     tst.w         (sp)+                      ; on character boundary?
     ble.s         @FixB                      ; yes, continue
     sub.l         #1,AInfo.StrPtr(A6)        ; adjust pointer
     add.w         #1,AInfo.StrCnt(A6)        ; adjust count
@FixB
     move.w        BStrLen(a6), d0            ; B length
     sub.w         BInfo.StrCnt(a6),d0        ; see if its changed
     beq.s         Quit Init                  ; B is done if not
     sub.l         #2,sp                      ; return param
     move.l        BStrText(a6), -(sp)        ; textBuf
     move.w        d0, -(sp)                  ; textOffset
     _CharByte
     tst.w         (sp)+                      ; on character boundary?
     ble.w         @QuitInit                  ; yes, continue
     sub.l         #1,BInfo.StrPtr(A6)        ; adjust pointer
     add.w         #1,BInfo.StrCnt(A6)        ; adjust count
@QuitInit
     bra.s         ReturnEQ                   ; return to the caller.
     EndWith
```

The Fetch Procedure

The Fetch Procedure is used to fetch a character from a string, updating the pointer and length to reflect the remainder of the string. For example, the following code changes the text comparison for Yugoslavian:

```
;---------------------------------------------------------------------
; Routine          FetchProc
; Input            A2          String Data Structure
;                  A3          String pointer (one past fetched char)
;                  A6          Local Frame
;                  D4.W        Character: top byte is fetched character, bottom
;                               is zero
;                  D5.B        1 if string is empty, otherwise 0
; Output           D4.W        Character: top byte set to character, bottom to
;                              extension
;                  D5.B        1 if string is empty, otherwise 0
; Trashes          Standard regs:              A0/A1/D0-D2
; Function         This routine returns the characters that are fetched from
;                  the string, if they are not just a sequence of single bytes.
;---------------------------------------------------------------------

FetchProc
     tst.b         d5                         ; more characters in string?
     bne.s         ReturnEq                   ; no -> bail out.

     move.w        d4,d0                      ; load high byte.
     move.b        (a3),d0                    ; load low byte.
```

```
     lea          pairTable,a1                ; load table address

@compareChar
     move.w       (a1)+,d1                    ; pair = 0?
     beq.s        ReturnEq                    ; yes -> end of table.
     cmp.w        d0,d1                       ; legal character pair?
     bne.s        @compareChar                ; no -> try the next one.
     add.w        #1,a3                       ; increment pointer.
     sub.w        #1,StrCnt(a2)               ; decrement length.
     addx.w       d5,d5                       ; empty -> set the flag.
     move.w       d0,d4                       ; copy character pair.
     rts                                      ; return to caller with CCR=NE

pairTable
     dc.b         'Lj'                        ; Lj
     dc.b         'LJ'                        ; LJ
     dc.b         'lJ'                        ; lJ
     dc.b         'lj'                        ; lj

     dc.b         'Nj'                        ; Nj
     dc.b         'NJ'                        ; NJ
     dc.b         'nJ'                        ; nJ
     dc.b         'nj'                        ; nj

     dc.b         'D', $be                    ; Dz-hat
     dc.b         'D', $ae                    ; DZ-hat
     dc.b         'd', $ae                    ; dZ-hat
     dc.b         'd', $be                    ; dz-hat

     DC.B         $00, $00                    ; table end
```

The same sort of procedure is used for Japanese or other double-byte scripts, in order to combine two bytes into a single character for comparison.

```
FetchProc
     with         IUStrData
     tst.b        d5                          ; empty string?
     bne.s        ReturnEq                    ; exit if length = 0

; if we have a double-byte char, add the second byte
     lea          CurChar(a2),a0              ; pass pointer
     move.w       d4,(a0)                     ; set value at ptr
     clr.w        d0                          ; pass length

     sub.l        #2,SP                       ; allocate return
     move.l       a0,-(sp)                    ; pointer
     move.w       d0,-(sp)                    ; offset
     _CharByte
     tst.w        (sp)+                       ; test return
     bmi.s        @DoubleByte                 ; skip if high byte (first two)

; we don't have a double byte, but two special cases combine second bytes
     move.b       (a3),d0                     ; get next byte
     cmp.b        #$DE,d0                     ; nigori?
     beq.s        @DoubleByte                 ; add in
     cmp.b        #$DF,d0                     ; maru?
```

```
     bne.s          ReturnEq                    ; exit: single byte

@DoubleByte
     move.b         (a3)+,d4                    ; get next byte
     subq.w         #1,StrCnt(A2)               ; dec string length
     addx.w         d5,d5                       ; set x=1 if string len = 0
     rts                                        ; return to caller with CCR=NE
```

The Project Procedure

The Project Procedure is used to find the primary ordering for a character. This routine will map characters that differ only in the secondary ordering onto a single character, typically the unmodified, uppercase character. For example, the following changes the comparison order for some Norwegian characters, so that they occur after 'Z.'

```
;---------------------------------------------------------------------
; Routine           ProjectProc
; Input             A2          String Data Structure
;                   D4.W        Character (top byte is char, bottom is extension
;                               (the extension is zero unless set by FetchProc))
; Output            D4.W        Projected Character
;                   CCR         NE to skip normal Project
; Trashes           Standard regs:              A0/A1/D0-D2
; Function          This routine projects the secondary characters onto primary
;                   characters.
;                   Example: a,ä,Ä -> A
;---------------------------------------------------------------------

ProjectProc
     lea            ProjTable,A1                ; load table address.
@findChar
     move.l         (a1)+,D0                    ; get entry
     cmp.w          d0,d4                       ; original ≥ entry?
     bhi.s          @findChar                   ; no, try the next entry.
     bne.s          ReturnEq                    ; not equal, process normally

@replaceChar
     swap           d0                          ; get replacement
     move.w         d0,d4                       ; set new character word.
@doneChar
     rts                                        ; CCR is NE to skip project.

ProjTable
;    Table contains entries of the form  r1, r2, o1, o2,
;    where r1,r2 are the replacement word, and
;    o1, o2 are the original character.
;    The entries are sorted by o1,o2 for use in the above algorithm

     DC.B           'Z', 3, 'Å', 0              ; Å after Ø
     DC.B           'Z', 3, 'å', 0              ; å after Ø
     DC.B           'Z', 1, 'Æ', 0              ; Æ after Z
     DC.B           'Z', 2, 'Ø', 0              ; Ø after Æ
     DC.B           'Z', 1, 'æ', 0              ; æ after Z
     DC.B           'Z', 2, 'ø', 0              ; ø after Æ
     DC.L           $FFFFFFFF                   ; table end
```

The Project procedure can also be used to undo the effects of the normal projection. For example, suppose that "œ" is not to be expanded into "oe": in that case, a simple test can be made against 'œ',0, returning `NE` if there is a match, so that the normal processing is not done. To expand one character into two, the routine should return the first replacement character in `D4.W`, and modify two fields in the `IUStrData` field. For example, given that `A1` points to a table entry of the form (primaryCharacter: Word; secondaryCharacters: Word), the following code could be used:

```
     …
     move.w        (a1)+,d4                  ; return first, primary character
     move.w        (a1)+,CurChar(A2)         ; original => first, modified
char.
     addq.b        #1,JustAfter(A2)          ; set to one (otherwise zero)
     move.b        (a1),BufChar(A2)          ; store second character (BYTE!)
     ...
```

`CurChar` is where the original character returned by `FetchChar` is stored. If characters are different even after being projected onto their respective primary characters, then the `CurChar` values for each string will be compared. `JustAfter` indicates that the expanded character should sort after the corresponding unexpanded form. This field must be set whenever `CurChar` is modified in order for the comparison to be fully ordered. `BufChar` stores the next byte to be retrieved from the string by `FetchChar`.

To handle the case where characters are ignored unless the two compared strings are otherwise equal, the `IgnChar` flag can be set. This can be used to handle characters such as the hyphen in English, or vowels in Arabic.

```
     …
     cmp.w         #hyphen,d0                ; is it a ignorable?
     seq           IgnChar(a2)               ; set whether or not
     …
```

The Vernier Procedure

The Vernier Procedure is used to make a final comparison among characters that have the same primary ordering. It is only needed if the `CurChar` values are not ordered properly. For example, according to the binary encoding, å < Ã. To change this ordering so that uppercase letters are before lowercase letters, Ã is mapped to $7F in normal comparison. Notice that only the characters in the secondary ordering are affected: Ã can be mapped onto Z, but not onto Ä, since that would cause a collision.

```
;---------------------------------------------------------------------
; Routine          VernierProc
; Input            D4.B        High byte of character
;                  D5.B        Low byte of character
; Output           D4.B        High byte of character
;                  D5.B        Low byte of character
;                  CCR         NE if to skip standard Vernier
; Trashes          Standard regs:              A0/A1/D0-D2
; Function         The Vernier routine compares characters within the secondary
;                   ordering if two strings are otherwise equal.
;                   Example: (a,A,Ä,ä)
;---------------------------------------------------------------------
```

```
VernierProc
     not.b          d4                          ; invert secondary ordering
     not.b          d5                          ; ditto for lower byte
     bra.s          ReturnEq                    ; normal processing afterwards
```

Installing an `itl2` resource

To write an `itl2` resource, follow the guidelines in Technical Note #110 for writing standalone code in MPW. The code should be written in assembly language, and it must follow the specifications given in this technical note or serious system errors could occur whenever string comparisons are made.

The default comparison routine is in the `itl2` resource of the System file. In order to use a comparison routine other than the standard one, you should include an `itl2` resource in your application with the same name and resource ID as the one in the System file that you wish to change. The Resource Manager will look for the resource in the application resource file before it looks in the System resource file, so your string comparison routine will be used instead of the default one.

It is generally a dangerous practice to change a system resource since other applications may depend on it, but if you have good reasons to permanently change the system `itl2` resource so that all applications use a different comparison routine, then you should write an installer script to change the `itl2` resource in the System resource file. Writing an installer script is documented in Technical Note #75. You are required to write an installer script if you are planning to ship your application on a licensed system software disk and your application makes a permanent change to any resources in the System file. We strongly discourage changing the System `itl2` as that would change the behavior of string comparison and sorting for all applications. If that is your intent, then you should write an installer script. However, if you are changing the `itl2` resource in the System file for academic or internal use, then you can use a resource editor such as ResEdit to copy your `itl2` resource into the System file.

#179: Setting ioNamePtr in File Manager Calls

See also: The File Manager

Written by: Jim Friedlander November 2, 1987
Updated: March 1, 1988

It is very important to set `ioNamePtr` when making `PB` calls, even if you don't want those calls to return a name. Whenever *Inside Macintosh* indicates that `ioNamePtr` is either required for input or returns something, you must set `ioNamePtr` to either `nil` (if you aren't using a name) or to point to storage for a `Str255`.

If you don't explicitly set `ioNamePtr`, strange and unusual crashes may occur, depending on the machine/configuration your code is run on.

Macintosh
Technical Notes

Developer Technical Support

#180: MultiFinder Miscellanea

Revised by: Dave Radcliffe & Keith Rollin
Written by: Jim Friedlander

August 1989
November 1987

This Technical Note discusses MultiFinder issues of which programmers should be aware.
Changes since June 1988: Updated and generalized sample code to reflect new MPW 3.0 calls in both C and Pascal for saving and restoring `A5` for interrupt code that accesses application globals. Removed text that can be found in *Programmer's Guide to MultiFinder*, and added a note about `_PostEvent`.

Switching

For conceptual clarity, it is best to think of MultiFinder 6.0 and earlier as using three types of switching: major, minor, and update. All switching occurs at a well defined times, namely, when a call is made to either `_WaitNextEvent`, `_GetNextEvent`, or `_EventAvail`.

Major switching is a complete context switch, that is, an application's windows are moved from the background to the foreground or vice versa. `A5` worlds are switched, and the application's low-memory world is switched. If the application accepts Suspend and Resume events, it is so notified at major switch time.

Major switching will not occur when a modal dialog is the frontmost window of the front layer, though minor and update switching will occur. To determine whether major switching will occur, MultiFinder checks (among other things) if the window definition procedure of that window is `dBoxProc`. If it is, then MultiFinder won't allow a switch via the user clicking on another application. A window definition procedure of `dBoxProc` is specifically reserved for modal dialogs—when most users see a `dBoxProc`, they are expecting a modal situation. If you are using a `dBoxProc` for a non-modal window, we strongly recommend that you change it to some other window type, or risk the wrath of the User–Interface Thought Police (UITP).

Minor switching occurs when an application needs to be switched out to give time to background processes. In a minor switch, `A5` worlds are switched, as are low-memory worlds, but the application's layer of windows is **not** switched, and the application won't be notified of the switch via Suspend and Resume events.

Update switching occurs when MultiFinder detects that one or more of the windows of an application that is not frontmost needs updating. This happens whether or not the application has the `canBackground` bit in the `'SIZE' -1` resource set. This switch is very similar to minor switching, except that update events are sent to the application whose window need updating.

Both minor and update switches should be transparent to the frontmost application.

Suspend and Resume Events

If your application does not accept Suspend and Resume events (as set in the 'SIZE' -1 resource), then if a mouse-click event occurs in a window that isn't yours, MultiFinder will send your application a mouse-down event with code inMenuBar (with menuID equal to the ID of the Apple menu and menuItem set to "About MultiFinder..."). The reason that MultiFinder does this is to force your application to think that a desk accessory is opening, so that it will convert any private scrap that it might be keeping. MultiFinder is expecting your application to call _MenuSelect—if you don't, it will currently issue a few more mouse-down events in the menu bar before finally giving up. This isn't really a problem, but a lot of developers have run into it, especially in "quick and dirty" applications.

If you are switching menu bars with _SetMenuBar (and switching the Apple Menu) during the execution of your application, then you should definitely make sure that your application accepts Suspend and Resume events. MultiFinder records the ID of the original Apple menu that you use and won't keep track of any changes that you make to the Apple menu. So, in the above situation, MultiFinder will give you a mouse-down event in the menu bar with the menuItem set to the item number of "About MultiFinder..." that was in the original Apple menu, which could be quite a confusing situation. If you set the MultiFinder friendly bit in the 'SIZE' resource, MultiFinder will never give you these mouse-down events.

Referencing Global Data (A5 and MultiFinder)

MultiFinder maintains a separate A5 world for each application. MultiFinder switches A5 worlds as appropriate, so most applications don't have to worry about A5 at all (except to make sure that it points to a valid QuickDraw global record at _GetNextEvent or _WaitNextEvent time). MultiFinder also switches low-memory globals for you. To get the value of the application's A5, use the routines from Technical Note #208, Setting and Restoring A5.

If an application uses routines that execute at interrupt time and accesses globals, then it needs to be concerned about A5. MultiFinder affects four basic types of interrupt routines:

- VBL tasks
- Completion routines
- Time Manager tasks
- Interrupt service routines

VBL Tasks

If an application installs a VBL task into its application heap, MultiFinder will currently "unhook" that VBL routine when it switches that application out (using either a major or a minor switch). It will "rehook" it when the application is switched back in. A VBL task that is installed in the system heap will always receive time, that is, it will never be "unhooked." Given this condition, it is technically not necessary for a VBL task that is in the application's heap to worry about its A5 context, since it will only be running when that application's partition is switched in. However, we would still like to encourage you to set up A5 by carrying its value around with the VBL, since we may change the way this works in future versions of MultiFinder (and even without MultiFinder, the VBL could trigger at a time when A5 is not correct).

The following short MPW examples show how to do this using the new MPW 3.0 calls mentioned in Technical Note #208. Please note that this technique does **not** involve writing into your code segment (we'll get to that later), we just put our value of the application's A5 in a location where

we can find it from our VBL task. Nor does it depend on the VBL task information being allocated globally. This gives you more flexibility setting up your VBL.

This example also serves to demonstrate how one might write a completion routine for an asynchronous Device Manager call. It is not intended to be a complete program, nor to demonstrate optimal techniques for displaying information.

MPW Pascal 3.0

```
UNIT   VBLS;

{$R-}

INTERFACE

USES
       Dialogs, Events, OSEvents, Retrace, Packages, Types, Traps;

CONST
       Interval = 6;
       rInfoDialog = 140;
       rStatTextItem = 1;

TYPE
       { Define a record to keep track of what we need.  Put theVBLTask into the
       record first because its address will be passed to our VBL task in A0. }
       VBLRec = RECORD
              theVBLTask:    VBLTask;              { the actual VBLTask }
              VBLA5:         LongInt;              { saved CurrentA5 where we can find it }
       END;
       VBLRecPtr = ^VBLRec;

VAR
       gCounter:      LongInt;                     { Global counter incremented by VBL }

PROCEDURE InstallVBL;

IMPLEMENTATION

{ GetVBLRec returns the address of the VBLRec associated with our VBL task.
  This works because on entry into the VBL task, A0 points to the theVBLTask
  field in the VBLRec record, which is the first field in the record and that
  is the address we return.  Note that this method works whether the VBLRec
  is allocated globally, in the heap (as long as the record is locked in
  memory) or if it is allocated on the stack as is the case in this example.
  In the latter case this is OK as long as the procedure which installed the
  task does not exit while the task is running.  This trick allows us to get
  to the saved A5, but it could also be used to get to anything we wanted to
  store in the record. }
FUNCTION GetVBLRec: VBLRecPtr;
       INLINE $2E88;                        { MOVE.L A0,(A7) }

PROCEDURE DoVBL (VRP: VBLRecPtr);
{ DoVBL is called only by StartVBL }
BEGIN
       gCounter := gCounter + 1;                          { Show we can set a global }
       VRP^.theVBLTask.vblCount := Interval; { Set ourselves to run again }
END;

PROCEDURE StartVBL;
{ This is the actual VBL task code.  It uses GetVBLRec to get our VBL record
  and properly set up A5.  Having done that, it calls DoVBL to increment a
  global counter and sets itself to run again.  Because of the vagaries of
  MPW C 3.0 optimization, it calls a separate routine to actually access
```

```
  global variables.  See Tech Note #208 for the reasons for this, as well
  as for a description of SetA5. }
VAR
        curA5:                  LongInt;
        recPtr:                 VBLRecPtr;

BEGIN
        recPtr := GetVBLRec;                            { First get our record }
        curA5:= SetA5(recPtr^.VBLA5);                   { Get our application's A5 }

                                                        { Now we can access globals }
        DoVBL (recPtr);                                 { Call another routine to do actual work }

        curA5:= SetA5(curA5);                           { restore original A5, ignoring result }
END;


PROCEDURE InstallVBL;
{ InstallVBL creates a dialog just to demonstrate that the global variable
  is being updated by the VBL Task.  Before installing the VBL, we store
  our A5 in the actual VBL Task record, using SetCurrentA5 described in
  Tech Note #208.  We'll run the VBL, showing the counter being incremented,
  until the mouse button is clicked.  Then we remove the VBL Task, close the
  dialog, and remove the mouse down events to prevent the application from
  being inadvertently switched by MultiFinder. }

VAR
        theVBLRec:              VBLRec;
        infoDPtr:               DialogPtr;
        infoDStorage:           DialogRecord;
        numStr:                 Str255;
        theErr:                 OSErr;
        theItemHandle:          Handle;
        theItemType:            INTEGER;
        theRect:                Rect;

BEGIN
        gCounter:= 0;                                   { initialize the global variable }
        infoDPtr:= GetNewDialog(rInfoDialog, @infoDStorage, Pointer(-1));
        DrawDialog(infoDPtr);
        GetDItem(infoDPtr, rStatTextItem, theItemType, theItemHandle, theRect);

        theVBLRec.VBLA5:= SetCurrentA5;                 { get our A5 }
        WITH theVBLRec.theVBLTask DO
                BEGIN
                        vblAddr:= @StartVBL;            { pointer to VBL code }
                        vblCount:= Interval;            { frequency of VBL in System ticks }
                        qType:= ORD(vType);             { qElement is a VBL type }
                        vblPhase:= 0;                   { no phases }
                END;

        theErr:= VInstall(@theVBLRec.theVBLTask);       { install this VBL task }
        IF theErr = noErr THEN                          { we'll show the global value in }
                BEGIN                                   { the dialog until a mouse click }
                        REPEAT
                                NumToString(gCounter, numStr);
                                SetIText(theItemHandle, numStr);
                        UNTIL Button;
                theErr:= VRemove(@theVBLRec.theVBLTask); { remove the VBL task }
                END;

        CloseDialog(infoDPtr);                          { get rid of the info dialog }
        FlushEvents(mDownMask, 0);                      { remove all mouse down events }
END;

END.
```

MPW C 3.0

```
#include <Events.h>
#include <OSEvents.h>
#include <OSUtils.h>
#include <Dialogs.h>
#include <Packages.h>
#include <Retrace.h>
#include <Traps.h>

#define INTERVAL            6
#define rInfoDialog         140
#define rStatTextItem       1

/*
 *  These are globals which will be referenced from our VBL Task
 */
long    gCounter;           /* Counter incremented each time our VBL gets called */

/*
 *  Define a struct to keep track of what we need.  Put theVBLTask into the
 *  struct first because its address will be passed to our VBL task in A0
 */
struct VBLRec {
        VBLTask         theVBLTask;     /* the VBL task itself */
        long            VBLA5;          /* saved CurrentA5 where we can find it */
};
typedef struct VBLRec VBLRec, *VBLRecPtr;

/*
 *      GetVBLRec returns the address of the VBLRec associated with our VBL task.
 *  This works because on entry into the VBL task, A0 points to the theVBLTask
 *  field in the VBLRec record, which is the first field in the record and that
 *  is the address we return.  Note that this method works whether the VBLRec
 *  is allocated globally, in the heap (as long as the record is locked in
 *  memory) or if it is allocated on the stack as is the case in this example.
 *  In the latter case this is OK as long as the procedure which installed the
 *  task does not exit while the task is running.  This trick allows us to get
 *  to the saved A5, but it could also be used to get to anything we wanted to
 *  store in the record.
  */
VBLRecPtr GetVBLRec ()
        = 0x2008;                   /* MOVE.L      A0,D0 */

/*
 *  DoVBL is called only by StartVBL ()
 */
void DoVBL (VRP)
VBLRecPtr       VRP;
{
        gCounter++;                                     /* Show we can set a global */
        VRP->theVBLTask.vblCount = INTERVAL;            /* Set ourselves to run again */
}

/*
 *  This is the actual VBL task code.  It uses GetVBLRec to get our VBL record
 *  and properly set up A5.  Having done that, it calls DoVBL to increment a
 *  global counter and sets itself to run again.  Because of the vagaries of
 *  MPW C 3.0 optimization, it calls a separate routine to actually access
 *  global variables.  See Tech Note #208  - "Setting and Restoring A5" for
 *  the reasons for this, as well as for a description of SetA5.
 */
void StartVBL ()
{
```

```
        long            curA5;
        VBLRecPtr       recPtr;

        recPtr = GetVBLRec ();              /* First get our record */
        curA5 = SetA5 (recPtr->VBLA5);      /* Get the saved A5 */

                                            /* Now we can access globals */
        DoVBL (recPtr);                     /* Call another routine to do actual work */

        (void) SetA5 (curA5);               /* Restore old A5 */
}

/*
 *  InstallVBL creates a dialog just to demonstrate that the global variable
 *  is being updated by the VBL Task.  Before installing the VBL, we store
 *  our A5 in the actual VBL Task record, using SetCurrentA5 described in
 *  Tech Note #208.  We'll run the VBL, showing the counter being incremented,
 *  until the mouse button is clicked.  Then we remove the VBL Task, close the
 *  dialog, and remove the mouse down events to prevent the application from
 *  being inadvertently switched by MultiFinder.
 */
void InstallCVBL ()
{
        VBLRec                  theVBLRec;
        DialogPtr               infoDPtr;
        DialogRecord            infoDStorage;
        Str255                  numStr;
        OSErr                   theErr;
        Handle                  theItemHandle;
        short                   theItemType;
        Rect                    theRect;

        gCounter = 0;                       /* Initialize our global counter */
        infoDPtr = GetNewDialog (rInfoDialog, (Ptr) &infoDStorage, (WindowPtr) -1);
        DrawDialog (infoDPtr);
        GetDItem (infoDPtr, rStatTextItem, &theItemType, &theItemHandle,
                &theRect);

        /*
         *  Store the current value of A5 in the MyA5 field.  For more
         *  information on SetCurrentA5, see Tech Note #208- "Setting and
         *  Restoring A5".
         */
        theVBLRec.VBLA5 = SetCurrentA5 ();
        /* Set the address of our routine */
        theVBLRec.theVBLTask.vblAddr = (VBLProcPtr) StartVBL;
        theVBLRec.theVBLTask.vblCount = INTERVAL;  /* Frequency of task, in ticks */
        theVBLRec.theVBLTask.qType = vType;        /* qElement is a VBL task */
        theVBLRec.theVBLTask.vblPhase = 0;

        /* Now install the VBL task */
        theErr = VInstall ((QElemPtr)&theVBLRec.theVBLTask);

        if (!theErr) {
                do {
                        NumToString (gCounter, numStr);
                        SetIText (theItemHandle, numStr);
                } while (!Button ());
                theErr = VRemove ((QElemPtr)&theVBLRec.theVBLTask); /* Remove it when done */
        }

        /* Finish up */
        CloseDialog (infoDPtr);                         /* Get rid of our dialog */
        FlushEvents (mDownMask, 0);                     /* Flush all mouse down events */
}
```

Completion Routines

Currently, MultiFinder will not do a major, minor, or update switch if an asynchronous File Manager call is pending. This may not be true in the future. We recommend that you use the above technique to save A5 for asynchronous File Manager calls. MultiFinder does allow a switch if an asynchronous Device Manager or Sound Manager call is pending. When the call completes, the completion routine has no way of knowing whose partition is active, that is, it doesn't know if A5 is valid (it needs A5 if it wants to access a global). Sounds pretty hopeless, huh?

Well, actually this one is quite easy, you just need to put the value of A5 that "belongs" to your partition in a place where you can find it from your completion routine. It is guaranteed that A0 will point to your parameter block when your completion routine is called, so you can use the same technique shown with VBL tasks to put the value of A5 at a known offset from the beginning of the parameter block, and then reference it from A0. Completion routines are normally written in assembly language, though you can also write them in a high-level language. A simple example of how to do this in MPW Pascal and C can be found in the previous section about VBL tasks (it was easier to provide a clear, concise example for VBL tasks than for asynchronous Device Manager completion routines).

Time Manager Tasks

The Time Manager was rewritten for System 6.0.3. The new version will put a pointer to the TMTask record in A1. This is not true in System 6.0.2 or earlier. The technique shown in the example VBL for accessing an application's globals is possible using System 6.0.3 and the Time Manager. Prior to System 6.0.3, the task must also store the application's A5 into its code. This method is not a very good idea and runs the risk of incompatibility (self–modifying code).

Interrupt Service Routines

If your application needs to get to its application globals, and it replaces the standard 68xxx interrupt vectors (levels 1-7) with pointers to its own routines, it must also store the application's A5 into its code (since there is no parameter block for interrupt service routines). This method is not a very good idea and runs the risk of compatibility (self–modifying code).

Note: WDEFs should also maintain a copy of A5 in the same fashion as Time Manager tasks (prior to System Software 6.0.3) and set up A5 when called; WDEFs should also be non-purgeable.

Launching and MultiFinder

Technical Note #126 discusses the sublaunching feature of Systems 4.1 and newer. If you are running MultiFinder, and you use the technique demonstrated in that Technical Note, your application will be able to launch the desired application and remain open. Note: MultiFinder does not support _Chain; your application should never call this trap.

The application that you launch will become the foreground application. Unlike non-MultiFinder systems, when the user quits the application that you have sublaunched, control will not necessarily return to your application, but rather to the next frontmost layer.

Note: The warnings in Technical Note #126 about sublaunching still apply, but, if you still wish to sublaunch, we strongly recommend that you set both high bits of LaunchFlags.

The Scrap and MultiFinder

MultiFinder 6.0 and earlier keeps separate scrap variables for each partition. MultiFinder only checks to see whether or not to increment the other partitions' `scrapCount` variables in response to a user-initiated Cut or Copy. To do this, it watches for a call to `_SysEdit (SystemEdit)` or a menu event to determine if an official Cut or Copy command has been issued.

When an application calls `_PutScrap` or `_ZeroScrap` in response to a Cut or Copy menu selection, the other partitions' `scrapCount` variables will be incremented (the other partitions will know that something new has been put in the scrap).

_UnmountVol and MultiFinder

`_UnmountVol` was changed in System 4.2 so that it would work better in a shared environment. In systems 4.1 and prior, `_UnmountVol` would successfully unmount a volume even if files were open on that volume. Under MultiFinder, that would be disastrous, since one application could unmount a volume that another application was using (this exact problem could occur when MultiFinder is not active, if a DA unmounted a volume "out from under" an application).

System 4.2 changes the behavior of `_UnmountVol` (whether or not MultiFinder is active) so that it returns a -47 (`FBsyErr`) error if any files are open on the volume you wish to unmount. Since the Finder always has a Desktop file open for each volume, a call to `_UnmountVol` asks it to close the Desktop file so you won't get an error if the only file open is the Desktop file. However, there is a bug with this new behavior. In System 6.0.3, and earlier, `_UnmountVol` does not close the Desktop file for MFS–formatted volumes. Only the Finder can unmount a MFS volume (when the user drags the disk icon to the trash).

Displaying a Splash Screen

Some applications like to put up a "splash screen" to give the user something to look at while the application is loading. If your application does this **and** has the `canBackground` bit set in the size resource, then it should call `_EventAvail` several times (or `_WaitNextEvent` or `_GetNextEvent`) before putting up the splash screen, or the splash screen will come up behind the frontmost layer. If the `canBackground` bit is set, MultiFinder will not move your layer to the front until you call `_GetNextEvent`, `_WaitNextEvent`, or `_EventAvail`.

The Apple Menu and MultiFinder

Applications should avoid doing anything untoward with the Apple menu. For example, if your application puts an icon next to the "About MyApplication..." item, MultiFinder may unceremoniously write over it. It is important to consider the Apple Menu owned by the system. You can have the standard about item, but other than this, you should avoid using the Apple menu. Don't make any assumptions about the contents of this menu. Even reading from its data may be a compatibility risk since its structure may change.

Interprocess Communication

MultiFinder 6.0, and earlier, does not have full-fledged interprocess communication facilities. There is no standard way to communicate between applications in MultiFinder 6.0. There are, however, a couple of ways to communicate between applications. For recommendations on attempting interprocess communication with MultiFinder 6.0, contact MacDTS at the address listed in Technical Note #0.

Note: It is in your best interest to wait until Apple implements Interapplication Communication (IAC) in System 7.0.

_PostEvent

Even though you can have many applications running at once, each with a fairly independent world, the Event Manager maintains only one event queue. Because of this single queue, and because there is no facility implemented to keep track of which events belong to which layer, all events in the queue are passed to the frontmost application. This situation can cause problems for applications that take advantage of application-defined events. If the application is in the background and posts one of these events, then it is the foreground application that receives it.

This does not apply to events which are not really stored in the event queue. The list of these events include, but is not limited to, activate and update events, which are generated by the Window Manager as needed, and are correctly routed to the right application.

Miscellaneous Miscellanea

The sound driver glue that shipped with MPW 1.0 and 2.0 is **not** MultiFinder compatible and should not be used. This also includes much of the glue supplied with older development systems. Instead, applications should be using the Sound Manager.

All code needs to be aware of the shared environment; this includes screen savers. Screen savers should make sure that background processing continues. A simple scenario for a screen saver that's an INIT might be: patch `_PostEvent` at INIT time, put up a full-screen black window spider, call `_WaitNextEvent`, and watch `_PostEvent` to see if an event that should cause the screen saver to go away has occurred.

Further Reference:

- *Inside Macintosh*, Volume V, Compatibility Guidelines
- *Programmer's Guide to MultiFinder* (APDA)
- Technical Note #126, Sub(Launching) from a High-Level Language
- Technical Note #129, _SysEnvirons: System 6.0 and Beyond
- Technical Note #158, Frequently Asked MultiFinder Questions
- Technical Note #205, MultiFinder Revisited: The 6.0 System Release
- Technical Note #208, Setting and Restoring A5

#181: Every Picture [Comment] Tells Its Story, Don't It?

See also: QuickDraw
Technical Note #91—
Optimizing for the LaserWriter—Picture Comments

Written by: Rick Blair
Updated:

November 2, 1987
March 1, 1988

Application-specific picture comment conflict and registration is addressed, along with Developer Technical Support's method for solving it.

I will assume that the nature and usefulness of picture comments are already well known. The problem I am addressing is that, as it stands, developers must register their comments with us (Developer Technical Support) or run the risk of using the same comments as those used by Apple or within another third party product.

The idea here is to provide a "metacomment" which will contain information about which application owns the comment as well as the comment itself:

```
ApplicationComment (long comment)                    kind = 100  size = n + 6
     data = application signature (i.e. 'MPNT' for MacPaint) = 4 bytes
            application "local" kind = 2 bytes
            comment data = n bytes
```

In this way each comment may be specific to an application. It is still up to a developer to publish information about the comments they have defined if they wish them to be understood and used by other programs.

Previously assigned and registered comments will still be valid. This means that those defined for the LaserWriter or MacDraw, for instance, will retain their normal meaning.

Suppose your application (creator = `'MYAP'`) wanted to define a comment. The appearance of the comment would be as follows (assuming you chose 128 to be the "local" kind for the comment):

```
kind = 100; data = 'MYAP' [4 bytes] + 128 [2 bytes] + additional data
```

Macintosh Technical Notes

#182: How to Construct Word-Break Tables

See also: The Script Manager

Written by: Mark Davis November 2, 1987
Updated: March 1, 1988

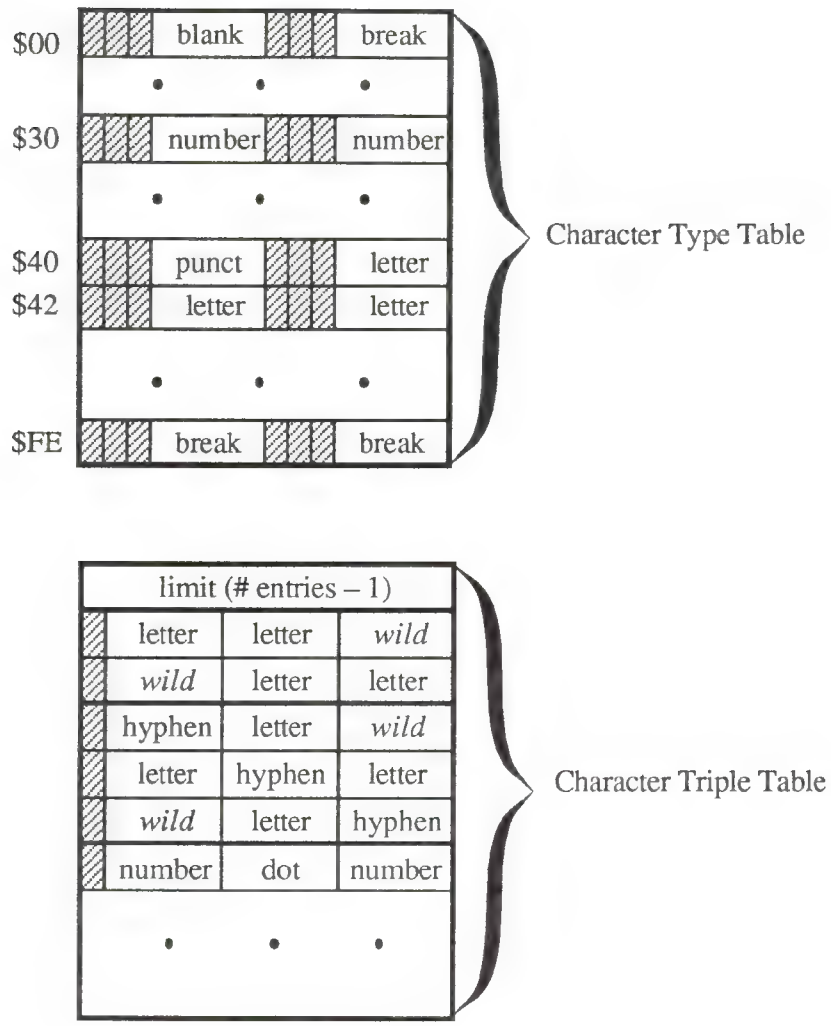
This technical note describes how to construct auxiliary break tables for use with the `FindWord` routine in the Script Manager.

Constructing break tables

The `FindWord` algorithm finds word boundaries by determining where words should not be broken. For example, "re-do" is one word: it should not be broken at the hyphen. In other words, a sequence of the form: (letter, hyphen, letter) should not be broken between the first and second or second and third character. This is called a continuation sequence. The algorithm used by the `FindWord` routine allows for continuation sequences of lengths one, two and three. Examples of a sequence of length two include (letter, letter), or (number, number). For a length of one, there is only one sequence, consisting of the characters of type `nonBreaking`: these characters are never separated from preceding or following characters.

For most scripts, this information about continuation sequences is packed into a table for use by the `FindWord` algorithm. (For complex scripts like Japanese, a different algorithm is used for portions of the script.) The default break tables for a given script can be overridden by a user-specified `breakTable` parameter, but should only be used for known scripts. That is, before overriding the `breakTable` parameter, the programmer should first check the script of the current font.

A break table consists of two sections, a 256 byte character type table followed by a character triple table.



The character type table is indexed by the character's ASCII code and contains one type value for each character. The character types in the table are limited to values between 1 and 31. There are two distinguishing values: the type `nonBreaking` (= 1) indicates that the character is non-breaking; it always continues a word. The type `wild` (=0) indicates that the character may or may not break, depending on information in the character triple table, as described below. Otherwise, the choice of numbers to represent character types is completely arbitrary.

For example, the following in MPW Assembler defines character types for use in a word-selection break table, then sets up a character type table using an assembly macro (`setByte`) to store character type values in an array. (Note that the character types could have been defined with equate definitions (`EQU`), rather than using the record structure.) Writing the setByte macro is left as an exercise to the reader. Note that the break value is the default. This value is not distinguished, but should have no continuation sequences.

```
;============================================================
charWordRec         record      0
wild                ds.b        1               ; constant! not in char table.
nonbreak            ds.b        1               ; constant! non-breaking space.
letter              ds.b        1               ; letters.
number              ds.b        1               ; digits.
break               ds.b        1               ; always breaks.
midLetter           ds.b        1               ; a'a.
midLetNum           ds.b        1               ; a'a 1'1.
preNum              ds.b        1               ; $, etc.
postNum             ds.b        1               ; %, etc.
midNum              ds.b        1               ; 1,1.
preMidNum           ds.b        1               ; .1234.
blank               ds.b        1               ; spaces and tabs.
cr                  ds.b        1               ; add carriage return
                    endr
;============================================================
     with                       charWordRec
wordTable
     dcb.b          256,break
     setByte        wordTable,nonBreak,$ca
     setByte        wordTable,letter,('A','Z'),('a','z')('Ä','ü')
     setByte        wordTable,letter,'Æ','Ø','æ','ø',('À','œ'),'ÿ'
     setByte        wordTable,midLetter,'-'
     setByte        wordTable,midLetNum,$27,''
     setByte        wordTable,number,('0','9')
     setByte        wordTable,preNum,'$','¢','£','¥'
     setByte        wordTable,postNum,'%'
     setByte        wordTable,midNum,','
     setByte        wordTable,preMidNum,'.'
     setByte        wordTable,blank,$00,' ',$09
     setByte        wordTable,cr,$0d
     endWith
;============================================================
```

The character triple table is a coded representation of a list of continuation sequences. It consists of a list of packed one word triples, preceded by a length word. This length word contains the number of triples minus one. Each triple contains three character types, either as derived from the `charType` table or the special type `wild` (= zero). The three types in a triple are packed into fields five bits apiece, with the most significant bit in the word cleared. The first type in the triple is the leftmost.

A continuation sequence of length three (xyz) is represented by entering three triples into the triple list: xyz, *xy, and yz* (where '*' stands for the type wild, which is always zero).



A continuation sequence of length two (xy) is represented by entering two triples into this list: *xy, and xy*. A continuation sequence of length one has no entry in the triple list: the character type is simply `nonBreaking`.



Note that the type `wild` cannot appear as the middle element of a triple. The words in the triple table must be sorted in ascending numerical order for future compatibility.

The following is an example of how a character triple table could be coded. The `defSeq` macro takes a continuation sequence as a parameter, and enters a set of triples into an internal array. The `dumpSeq` macro sorts the triples, and stores them in the proper order with `dc.w` commands. Once again, writing the macros `defSeq` and `dumpSeq` is left as an exercise for the reader.

```
;=============================================================
     with          charWordRec
     defSeq        letter,letter
     defSeq        letter,preMidNum,letter
     defSeq        letter,midLetter,letter
     defSeq        letter,midLetNum,letter

     defSeq        number,number
     defSeq        number,letter
     defSeq        number,midNum,number
     defSeq        number,midLetNum,number
     defSeq        number,preMidNum,number
     defSeq        number,postNum
     defSeq        preNum,number
     defSeq        preMidNum,number

     defSeq        blank,blank
     defSeq        blank,cr
     endWith

;=============================================================
     dc.w          ((wordEnd-wordBegin)/2)-1    ; length word.
wordBegin
     dumpSeq
wordEnd
;=============================================================
```

A series of blanks should generally select as a single word. Make certain, however, that a carriage return does not continue a word to the right (note how it has a separate character type from blank for this reason), otherwise word selection and wrapping do not work properly across paragraphs.

Extensions

The values 16-31 in the character type table entry for null ($00) (the first byte in the character type table) are reserved by Apple for future expansion. The use of one of these values indicates the presence of a supplementary table after the triple table.

#183: Position-Independent PostScript

See also: The Print Manager
QuickDraw
LaserWriter Reference Manual
Technical Note #91—
Optimizing for the LaserWriter—PicComments
PostScript Language Reference Manual, Adobe Systems
PostScript Language Cookbook and Tutorial, Adobe Systems

Written by: Scott "ZZ" Zimmerman November 2, 1987
Updated: March 1, 1988

This technical note describes a method for inserting position-independent PostScript into QuickDraw pictures.

There is a problem with pictures that contain PostScript code. Sometimes the PostScript code that is inserted into the picture is dependent on the position of the picture on the page. The problem arises when these pictures are cut or copied from their original position, and pasted into another position or even into another document. The PostScript code will not know the new location of the picture, and will not execute correctly.

The solution for this problem, is to provide some way for the PostScript code to determine the current location of the picture relative to the page that it is being printed on. This is done by inserting QuickDraw calls to position the LaserWriter's pen before inserting the position dependent PostScript code. When the PostScript in the picture is executed, the LaserWriter's pen location will be in a location relative to the position of the picture on the page.

The following example illustrates a method for positioning the LaserWriter's pen before inserting any PostScript code into the picture. The method uses QuickDraw calls to position the LaserWriter's pen, and will work with any application that supports QuickDraw pictures. Applications do not have to be changed to be able to print pictures which use this technique; normal calls to `DrawPicture` will work.

The following code fragment will create a picture that contains PostScript to draw a rectangle around the picture's frame:

```
FUNCTION CreatePicture(pictureRect: Rect): PicHandle;
    CONST
          PostScriptBegin = 190;
          PostScriptEnd = 191;
          PostScriptHandle = 192;
    VAR
          PSString    : Str255;
          PSHandle    : Handle;
          theError    : OSErr;
    BEGIN
          (* Create a new Picture. *)
          CreatePicture := OpenPicture(pictureRect);
          ClipRect(pictureRect);

          (* Set the pen size to 0,0 so the following Line will not *)
          (* be shown, it is only sent to position the pen.*)
          PenSize(0,0);

          (* Move the QuickDraw pen to the first pixel inside the *)
          (* the picture's frame.  This by itself will not *)
          (* change the LaserWriter's pen location! *)
          MoveTo(pictureRect.left, pictureRect.top);

          (* Force the LaserWriter's pen location to match the  *)
          (* QuickDraw pen location.  Actually any drawing command, *)
          (* such as LineTo or even DrawString will cause the *)
          (* LaserWriter's pen location to change.  This call to *)
          (* the Line procedure only causes the coordinates of *)
          (* the above MoveTo to be flushed to the LaserWriter. *)
          (* Because of the PenSize call above, no Line is drawn. *)
          Line(0,0);

          (* Reset the pen to its default size. *)
          PenSize(1,1);

          (* The LaserWriter's pen location has now been changed.  The *)
          (* PostScript currentpoint operator will now return the *)
          (* location of the center pixel of the Picture. *)
          (* Get the PostScript ready to be sent. *)
          (* currentpoint          - push the current Point onto the stack. *)
          (* newpath               - Begin a new Line segment. *)
          (* MoveTo                - Move to the currentpoint we saved above. *)
          (* nn nn rlineto         - frame the Picture with lines. *)
          (* stroke                - Draw the frame. *)
          PSString :=
          '100 0 rlineto 0 100 rlineto -100 0 rlineto 0 -100 rlineto stroke '
                    ;
          PSString[length(PSString)] := CHR(13); (* Don't forget CR. *)
          theError:=PtrToHand(Ptr(ORD4(@PSString)+1),
                              PSHandle,length(PSString));
          IF theError <> noErr THEN HandleError;
```

```
        (* Send the PostScript code to the LaserWriter. *)
        PicComment(PostScriptBegin,0,nil);

              (* QuickDraw calls made between the PostScriptBegin *)
              (* and PostScriptEnd PicComments will be ignored by *)
              (* devices that support PostScript. *)

              PicComment(PostScriptHandle,GetHandleSize(PSHandle),PSHandle);

        PicComment(PostScriptEnd,0,nil);

        (* Kill off the Handle we created and close the picture. *)
        DisposHandle(PSHandle);
        ClosePicture;
END; (* CreatePicture *)
```

See the LaserWriter Reference Manual for more information about PicComments. See the *PostScript Language Reference Manual* for more information about the `currentpoint` operator.

There are some important guidelines to follow when sending PostScript directly to the LaserWriter. See the PostScript Commands section of Technical Note #91, for a complete description of these guidelines.

Macintosh
Technical Notes

Developer Technical Support

#184: Notification Manager

Revised by: Rich Collyer
Written by: Darin Adler

December 1989
April 1988

This Technical Note describes the Notification Manager, the part of the operating system that lets an application, desk accessory, or driver alert the user.
Changes since October 1989: Clarified the section on error handling for calls to `_NMInstall`.

The Notification Manager, in System Software version 6.0 and later, provides the user with an asynchronous "notification" service. The Notification Manager is especially useful for background applications; the PrintMonitor application and the system alarm (set by the Alarm Clock desk accessory) both use its services.

Each application, desk accessory, or device driver can queue any number of notifications. For this reason, you should try to avoid posting multiple notifications since each one will be presented separately to the user (i.e., "you have mail," "you have mail," etc.).

The Notification Manager queue contains information describing each notification request; you supply a pointer to a queue element describing the type of notification you desire. The Notification Manager queue is a standard Macintosh queue, as described in the Operating System Utilities chapter of *Inside Macintosh*, Volume II-367. Each entry in the Notification Manager queue has the following structure:

```
TYPE NMRec = RECORD
               qLink:        QElemPtr;     {next queue entry}
               qType:        INTEGER;      {queue type -- ORD(nmType) = 8}
               nmFlags:      INTEGER;      {reserved}
               nmPrivate:    LONGINT;      {reserved}
               nmReserved:   INTEGER;      {reserved}
               nmMark:       INTEGER;      {item to mark in Apple menu}
               nmSIcon:      Handle;       {handle to small icon}
               nmSound:      Handle;       {handle to sound record}
               nmStr:        StringPtr;    {string to appear in alert}
               nmResp:       ProcPtr;      {pointer to response routine}
               nmRefCon:     LONGINT;      {for application use}
             END;
```

To use the Notification Manager, you must also use `_SysEnvirons` (discussed in *Inside Macintosh*, Volume V and Technical Note #129) to test the System Software version. If the System Software is not current and the Notification Manager routines are not present, display an alert to inform the user that your application requires System Software version 6.0 or newer, then exit.

Using the Notification Manager

Your program can request three types of notification:

- polite notification: a small icon that periodically appears in rotation with the Apple in the menu bar;
- audible notification: a sound to be played by the Sound Manager;
- alert notification: an dialog box containing a short message.

In addition, you can place a diamond mark next to the name of the requesting desk accessory or application in the Apple menu.

After you have notified the user as you feel necessary (placed a diamond mark in the Apple menu, added a small icon to the list of icons which rotate with the Apple in the menu bar, played a sound, and presented the user with a dialog box to acknowledge), you should call a response procedure.

How the Notification Manager Handles Notifications

When the Notification Manager handles a notification, it does one or more of the following (in this order):

- puts a diamond mark next to the requesting application or desk accessory in the Apple menu
- adds a small icon to the list of icons which rotate with the Apple in the menu bar
- plays a specified sound
- presents a dialog box for the user to acknowledge and dismiss
- calls the response procedure

At this point, the diamond mark in the Apple menu and the icon rotating with the Apple in the menu bar remain until your application removes the notification request from the queue. The Notification Manager only presents the sound and dialog box once.

Creating a Notification Request

To create a notification request, you must set up an `NMRec` with all the information about the notification you want:

`nmMark` contains 0 to not place a mark in the Apple menu, 1 to mark the current application, or the `refNum` of a desk accessory to mark that desk accessory. An application should pass 1, a desk accessory should pass its own `refNum`, and a VBL task should pass 0.

`nmSIcon` contains `NIL` for no icon in the menu bar, or a handle to a small icon to rotate with the Apple. (A small icon is a 16 x 16 bitmap, often stored in an `'SICN'` resource.) This handle does not need to be locked, but it must be non-purgeable.

`nmSound` contains `NIL` to use no sound, –1 to use the system beep sound, or a handle to a sound record which can be played with `_SndPlay`. This handle does not need to be locked, but it must be non-purgeable.

`nmStr` contains `NIL` for no alert, or a pointer to the string to appear in the alert.

nmResp contains NIL for no response procedure, –1 for a predefined procedure that removes the request immediately after it is completed, or a pointer to a procedure which takes one parameter, a pointer to your queue element. For example, the following is how you would declare it if it were named MyResponse:

```
PROCEDURE MyResponse (nmReqPtr: QElemPtr);
pascal void MyResponse (QElemPtr nmReqPtr);
```

Note that when the Notification Manager calls your response procedure, it does not set up A5 and low-memory globals for you. If you need to access your application's globals, you should save your application's A5 in the nmRefCon field as discussed later in this Note.

Response procedures should never draw or do "user interface" things. You should wait until the user brings the application or desk accessory to the front before responding to the user. Some good ways to use the response procedure are to dequeue and deallocate your Notification Manager queue element or to set an application global (being careful about A5), so the application knows when the user receives the notification.

You should probably use an nmResp of –1 with audible and alert notifications to remove the notification as soon as the sound has finished or the user has dismissed the dialog. You should not use an nmResp of –1 with an nmMark or an nmSIcon, because the Notification Manager would remove the diamond mark or small icon before the user could see it. Note that an nmResp of –1 does not deallocate the memory block containing the queue element, it only removes it from the notification queue.

You can also use nmRefCon; one convenient use is putting your application's A5 in this field so the response procedure can access application globals. For more information about this technique, refer to the section about VBL tasks in Technical Note #180.

Notification Manager Routines

The system automatically initializes the Notification Manager when it boots. To add a notification request to the notification queue, call _NMInstall. When your application no longer wants a notification to continue, it can remove the request by calling _NMRemove. Neither _NMInstall nor _NMRemove move or purge memory, and you can call either of them from completion routines or interrupt handlers, as well as from the main body of an application and the response procedure of a notification request.

```
FUNCTION NMInstall (nmReqPtr: QElemPtr) : OSErr;

      Trap macro     _NMInstall ($A05E)
      On entry       A0: theNMRec (pointer)
      On exit        D0: result code (word)
```

_NMInstall adds the notification request specified by nmReqPtr to the notification queue and returns one of the following result codes:

```
      Result codes   noErr              No error
                     nmTypErr (-299)    qType field is not ORD(nmType)
```

```
FUNCTION NMRemove (nmReqPtr: QElemPtr) : OSErr;

      Trap macro     _NMRemove ($A05F)
      On entry       A0: theNMRec (pointer)
      On exit        D0: result code (word)
```

_NMRemove removes the notification identified by nmReqPtr from the notification queue and returns one of the following result codes:

```
      Result codes   noErr              No error
                     qErr               Not in queue
                     nmTypErr (-299)    qType field is not ORD(nmType)
```

How to Call _NMInstall and _NMRemove

If you do not yet have glue for _NMInstall and _NMRemove, you can use the following from MPW (these are in the include files for MPW 3.0):

Pascal

```
FUNCTION NMInstall (nmReqPtr: QElemPtr) : OSErr;
 INLINE $205F, $A05E, $3E80;

 FUNCTION NMRemove (nmReqPtr: QElemPtr) : OSErr;
 INLINE $205F, $A05F, $3E80;
```

C

```
pascal OSErr NMInstall (QElemPtr nmReqPtr)
       = {0x205F, 0xA05E, 0x3E80};

pascal OSErr NMRemove (QElemPtr nmReqPtr)
       = {0x205F, 0xA05F, 0x3E80};
```

Also note that qType must be set to ORD(nmType), which is 8.

The following short code segments demonstrate the use of the Notification Manager in MPW C:

```
#include <OSUtils.h>
#include <Notification.h>

struct NMRec   myNote;        /* declare your NMRec */
Handle         ManDoneS;      /* declare a handle for the sound */
OSErr          err;           /* declare for err handling */

myNote.qType = nmType;        /* queue type -- nmType = 8 */
myNote.nmMark = 1;            /* get mark in Apple menu */
myNote.nmSIcon = nil;         /* no flashing Icon */

/* get the sound you want out of your resources */
ManDoneS = GetResource('snd ', soundID);

myNote.nmSound = ManDoneS;    /* set the sound to be played
myNote.nmStr = nil;           /* no alert box */
myNote.nmResp = nil;          /* no response procedure */
myNote.nmRefCon = nil;        /* set to nil since I don't need my A5*/
```

Before calling `_NMInstall`, you need to see if your application is running in the background. If your application is in the foreground, you really do not need to notify the user, but if your application is in the background, you should make the following call to install the notification event:

```
err = NMInstall ((QElemPtr) &myNote);
```

If the call to `_NMInstall` returns an error, you cannot install the notification event and must wait for the user to switch your application to the foreground before proceeding with anything else. While you are waiting for a resume event, you should take care of other events, such as updates. If you were able to install the notification, then you want to make sure to remove it when you are switched back into the foreground. The following code does just that:

```
err = NMRemove ((QElemPtr) &myNote);
```

Further Reference:

- *Inside Macintosh*, Volume II-367, V-591, The Operating System Utilities
- Technical Note #129, _SysEnvirons: System 6.0 and Beyond
- Technical Note #180, MultiFinder Miscellanea

#185: OpenRFPerm: What your mother never told you.

See also: The Resource Manager

Written by: Dave Burnard April 2, 1988

This note corrects an error in the description of the Resource Manager routine `OpenRFPerm` found in *Inside Macintosh Volume IV*.

On page IV-17 in the Resource Manager chapter of *Inside Macintosh Volume IV* it states,

> "`OpenRFPerm`, like `OpenResFile`, will not open the specified file twice; it simply returns the reference number already assigned to the file. In other words, `OpenRFPerm` cannot be used to open a second access path to a resource file..."

This statement is incorrect. `OpenRFPerm` behaves exactly as described if an application attempts to open a second access path, **with write access**, to a resource file without MultiFinder. With MultiFinder, however, if the second attempt comes from a different application than the one that originally opened the file `OpenRFPerm` **will not** return the reference number already assigned to the file, instead it will return –1 and `ResError` will be set to `opWrErr` (–49). In fact in similar circumstances with MultiFinder, `OpenResFile` behaves the same way, returning –1 and setting `ResError` to –49.

Note, however, that with or without MultiFinder, `OpenRFPerm` **will** create multiple, unique, read-only access paths to a resource file. Using this feature is not safe and should be avoided since if a resource file is opened twice, once with read-write permission and once with read-only permission then two copies of the resource map will exist in memory. If one of the resources in the file is changed and written to disk then the two maps will become inconsistent and the second access path will see what it thinks is a trashed resource file.

If you must use this technique for read-only access, only call `OpenRFPerm` when your application is ready to read information from the file and close the file immediately afterwards. Otherwise you risk having the resource file change out from under your application.

Macintosh
Technical Notes

Developer Technical Support

#186: Lock, Unlock the Range

Revised by: Jim Luther
Written by: Dave Burnard

February 1991
April 1988

This Technical Note discusses the _PBLockRange and _PBUnlockRange routines; how they act on local and shared volumes and why you should not set the ioPosMode field to fsFromLEOF in the parameter block for those routines when accessing a file on an AppleShare volume.

Changes since April 1988: Added information on how _PBLockRange and _PBUnlockRange really work.

A file can be opened with a shared read and write permission (ioPermssn = fsRdWrShPerm) to allow several users to share the data in that file. (When a file is opened more than once, each open is considered a different "user".) When you, as a user of a shared file, need to modify a portion of the file (for example, a record in a database), it is usually very desirable to make that portion of the file unavailable to other users while you are making your changes. The _PBLockRange call is provided to lock a portion or range of bytes of the file. _PBUnlockRange can be called later to unlock the portion of a file locked with _PBLockRange. If you have read the File Manager chapters of Inside Macintosh IV and V, you know all of that, but there are a few things that it doesn't tell you, and that is the topic of this Note.

Local Versus AppleShare

_PBLockRange and _PBUnlockRange do nothing when used on files that are on a local HFS (or MFS) volume unless local file sharing is in effect under System Software 7.0. The two routines do not return any errors and they do not lock or unlock a range of bytes. Following is a way to verify that _PBLockRange works on an open file:

```
{Start with an open file}
WITH theParmBlk DO
  BEGIN
    {ioRefNum from open call}
    ioCompletion := NIL;
    ioReqCount := 1;          {lock a single byte}
    ioPosMode := fsFromStart; {at the beginning of the file}
    ioPosOffset := 0;
  END;
theErr := PBLockRange(@theParmBlk, FALSE);   {lock the byte - ignore error}
theErr := PBLockRange(@theParmBlk, FALSE);   {lock it again}
IF theErr = afpRangeOverlap {see if PBLockRange locked it the first time}
  THEN
    BEGIN
      RangeLockPresent := TRUE; {Range was locked}
      theErr := PBUnlockRange(@theParmBlk, FALSE); {unlock the locked byte}
    END;
  ELSE RangeLockPresent := FALSE; {Range was not locked}
```

Luckily, _PBLockRange and _PBUnlockRange work when used on files that are on a shared volume (i.e., AppleShare) where the possibility of shared access usually comes up much more often. When _PBLockRange and _PBUnlockRange are used on a file on a shared volume, both calls are translated into an AppleTalk Filing Protocol (AFP) call, FPByteRangeLock. The FPByteRangeLock call locks or unlocks a specified range of bytes within an open fork.

Note: With System Software 7.0, file sharing can be started or stopped with the Sharing Setup Control Panel while your application is running. If file sharing is stopped, AppleTalk AFP specific attributes on local volumes, including locked ranges, become invalid. The bHasPersonalAccessPrivileges bit in the vMAttrib longword returned by _PBHGetVolParms can be used to check the current state of file sharing on local volumes.

One Way To Lock, Two Ways To Unlock

The File Manager only gives one way to lock a range of bytes of a file, the _PBLockRange routine. However, there are two ways to unlock a locked range of bytes of a file. The obvious method of unlocking a range of bytes is the _PBUnlockRange routine; the not so obvious method is to close the file. When a file is closed, all locked ranges held by a user of the file are unlocked.

The range of bytes unlocked by _PBUnlockRange must be the exact same range locked by a _PBLockRange call; you cannot unlock a portion of a locked range nor can you unlock portions of a file that are not locked. If, for some reason, you need to unlock a range you have locked and you do not know where the range started or how long the range is, you must close the file to unlock the range.

What Range Did I Just Lock?

A range of bytes may be locked relative to the beginning or the end of the file, however AFP's FPByteRangeLock call only lets you unlock a range of bytes relative to the beginning of the file. If a range of bytes is successfully locked with the FPByteRangeLock call, then a reply packet containing RangeStart, the number of the first byte of the range just locked (i.e., the offset from the beginning of the fork) is returned to FPByteRangeLock. That offset allows an application to set a lock when the application does not know the exact end of file (which can often happen when sharing a file) and later know what range to unlock.

This brings us to an interesting fact: _PBLockRange does not return the number of the first byte of the range just locked to your application. The reply packet containing RangeStart is returned by FPByteRangeLock, but _PBLockRange does not return it to your application. That would make it impossible to really know what range you just locked if the range locked were relative to the end of file. However, the Macintosh does not use the file server's end of file. Instead, the file's logical end of file is retrieved from the server when the file is opened and that local copy of end of file is used whenever the ioPosMode field is set to fsFromLEOF (the current logical end of file can be retrieved with _PBGetEOF).

An Off By EOF Bug

With Macintosh System Software 6.0.7 and earlier, the `_PBLockRange` call's parameters are incorrectly translated to the parameters passed to the AFP `FPByteRangeLock` call. When `ioPosMode` field is set to `fsFromLEOF` the offset parameter sent is equal to the offset supplied to the `_PBLockRange` call plus what your system locally thinks is the current EOF. This would work correctly except that the `Start/EndFlag` in the `FPByteRangeLock` call's parameter list is set to `End` (relative to end of file) which means that the range locked is EOF bytes further out than that for which you asked.

The problem can be avoided under System Software 6.0.7 and earlier by using `fsFromStart`, `fsAtMark`, or `fsFromMark` for the `ioPosMode` field. System Software 7.0 fixes the bug by setting the `Start/EndFlag` in the `FPByteRangeLock` call's parameter list to `Start` (relative to beginning of file) when `ioPosMode` is set to `fsFromLEOF`.

Conclusion

Remember, `_PBLockRange` does not do anything on local unshared volumes, and if you use `_PBLockRange` under Macintosh System Software 6.0.7 or earlier, never set `ioPosMode` to `fsFromLEOF`.

Further Reference:

- *Inside Macintosh*, Volumes IV & VI, The File Manager
- *Inside Macintosh*, Volume V, File Manager Extensions In a Shared Environment
- *Inside AppleTalk*, AppleTalk Filing Protocol

#187: Don't Look at ioPosOffset

See also: The Device Manager

Written by: Darin Adler April 2, 1988

The Device Manager chapter of *Inside Macintosh Volume II* says that `ioPosOffset` is passed to **and returned by** `Read` and `Write` calls. It also says that "After the read [or write] is completed, the position is returned in `ioPosOffset`..." Actually, `ioPosOffset` is not changed by either call.

Also note that device drivers should **only** look at the `dCtlPosition` field of the DCE, and should not look directly at the `ioPosOffset` field of the parameter block. The Device Manager sets up `dCtlPosition` for the driver, taking into account both the `ioPosMode` and the `ioPosOffset`.

#188: ChangedResource: Too much of a good thing.

See also: The Resource Manager

Written by: Dave Burnard April 2, 1988

The toolbox trap `ChangedResource` is used to inform the Resource Manager that the contents of a resource have changed and should be written to disk. The actual write occurs on the next call to `WriteResource` (for the specific resource) or `UpdateResFile` (for the resource file containing the specified resource). When called, `ChangedResource` reserves enough disk space to contain the changed resource. A little-known "feature" of `ChangedResource` is that it reserves disk space every time it is called. Thus if you call `ChangedResource` ten times on a large resource before the resource is actually written out, you may unexpectedly run out of disk space since ten times the amount of disk space actually needed will be reserved.

If your program frequently changes the contents of resources, especially large ones, then you should call `WriteResource` or `UpdateResFile` immediately after calling `ChangedResource`. Once the resource is actually written, the file's EOF will be set correctly, and the first subsequent call to `ChangedResource` will work correctly.

Macintosh
Technical Notes



Developer Technical Support

#189: Version Territory

Revised by: Rich Collyer
Written by: Darin Adler

October 1990
April 1988

This Technical Note describes the `'vers'` resource supported by Finder 6.1 and later.
Changes since April 1989: Changed MPW C code to reflect the changes in MPW C 3.1.

Finder 6.1 introduced a feature allowing the creator of a file to identify the version of that file as well as the version of a set of files which includes that file. These version numbers are stored in `'vers'` resources, and each contains a BCD form of the version number and a longer version message (which the Finder displays in the Get Info window for each file).

Apple's Version Numbering Scheme

Apple uses a version numbering scheme for its software products which you might want to adopt. Table 1 summarizes the scheme, which involves three numbers, each separated by periods.

Event	Version
First released version	1.0
First revision	1.1
First bug fix to the first revision	1.1.1
First major revision or rewrite	2.0

Table 1–Apple's Version Numbering Scheme

Note that Apple increments the first number when it releases a major revision, the second number when it releases a minor revision, and the third number when it releases a version to address bugs (the third number is omitted if it is zero).

During product development, Apple uses a version number followed by a suffix which indicates the stage of development. Table 2 presents a few examples.

Event	Version	Stage
First versions	1.0d1, 1.0d2...	development
Product features defined (begin testing)	1.0a1, 1.0a2...	alpha
Product is stable (begin final testing)	1.0b1, 1.0b2...	beta
First released version	1.0	release
First revision	1.1d1...1.1	
First bug fix to the first revision	1.1.1d1...1.1.1	
First major revision	2.0d1...2.0	

Table 2–Development Version Numbering

Version Resources

Each `'vers'` resource has the following format (described with a Rez template):

```
#include "SysTypes.r" /* for country codes */

type 'vers' {
    byte;              /* first part of version number in BCD */
    byte;              /* second and third parts of version number */
    byte development=0x20, alpha=0x40, beta=0x60, release=0x80;
    byte;              /* stage of non-release version */
    integer Country;   /* country code as in international utilities */
    pstring;           /* short version number */
    pstring;           /* long version message */
};
```

The short version number is a string which only contains the version number (e.g., "1.0"). The long version message can also include a copyright notice, a release date, or other information, but it should **not** include the name of the program. The following examples illustrate the proper use of the Rez template to create version resources:

```
resource 'vers' (1) {
    0x01, 0x00, release, 0x00, verUS,
    "1.0",
    "1.0 (US), ©1989 Inside Joke"
};
resource 'vers' (2) {
    0x12, 0x00, release, 0x00, verUS,
    "12.0",
    "Watt-R-Utilities Disk 12.0"
};

resource 'vers' (1) {
    0x23, 0x45, beta, 0x67, verFinland,
    "23.4.5b67",
    "23.4.5b67 (Finland), ©1989 Squid, Inc."
};
resource 'vers' (2) {
    0x55, 0x00, development, 0x67, verFinland,
    "55.0d67",
    "Friends of Skippy White 55.0d67"
};
```

The following is a type definition for `'vers'` resources in MPW Pascal:

```
NumVersion = PACKED RECORD
    CASE INTEGER OF
      0:
        (majorRev: SignedByte;  {1st part of version number in BCD}
        minorRev: 0..9;         {2nd part is 1 nibble in BCD}
        bugFixRev: 0..9;        {3rd part is 1 nibble in BCD}
        stage: SignedByte;      {stage code: dev, alpha, beta, final}
        nonRelRev: SignedByte); {revision level of non-released version}
      1:
        (version: LONGINT);     {to use all 4 fields at one time}
    END;
```

```
{ Numeric version part of 'vers' resource }
VersRecPtr = ^VersRec;
VersRecHndl = ^VersRecPtr;
VersRec = RECORD
    numericVersion: NumVersion; {encoded version number}
    countryCode: INTEGER;       {country code from intl utilities}
    shortVersion: Str255;       {version number string - worst case}
    reserved: Str255;           {longMessage string packed after shortVersion}
    END;
```

The type definition in MPW C is as follows (these structures are not needed in your code since they are included in the header file Files.h):

```
struct NumVersion {
    unsigned char majorRev;     /*1st part of version number in BCD*/
    unsigned int minorRev : 4;  /*2nd part is 1 nibble in BCD*/
    unsigned int bugFixRev : 4; /*3rd part is 1 nibble in BCD*/
    unsigned char stage;        /*stage code: dev, alpha, beta, final*/
    unsigned char nonRelRev;    /*revision level of non-released version*/
};

/* Numeric version part of 'vers' resource */
struct VersRec {
    NumVersion numericVersion;  /*encoded version number*/
    short countryCode;          /*country code from intl utilities*/
    Str255 shortVersion;        /*version number string - worst case*/
    Str255 reserved;            /*longMessage string packed after shortVersion*/
};

typedef VersRec *VersRecPtr, **VersRecHndl;
```

The `longMessage` string is not necessarily word-aligned due to the way the resource is formatted, so you should use `_BlockMove` to extract it from the record. Following are examples of this technique:

MPW Pascal

```
VAR
    version: VersRecHandle;
    messagePtr: StringPtr;
    longMessage: Str255;

version := GetResource ('vers', 1);
WITH version^^ DO
    BEGIN
    {calculate a pointer to the long message}
    messagePtr := StringPtr(Ord(@shortVersion[1])+Length(shortVersion));
    {move the long message into a string}
    BlockMove(Ptr(messagePtr), @longMessage, Length(messagePtr^)+1);
    END;
```

MPW C

```
VersRecHndl  version;
StringPtr    messagePtr;
char         *shortversion, *longversion;

version = (VersRecHndl) GetResource ('vers', 1);
/* calculate a pointer to the long message */
messagePtr = (StringPtr) (((unsigned long) &(**version).shortVersion[1]) +
             ((**version).shortVersion[0]));
/* move the long message into a string */
BlockMove(messagePtr, &longMessage, ((unsigned char) &messagePtr) + 1);
```

A file can contain either one, two, or no `'vers'` resources. If present, a `'vers' (1)` resource identifies the file version while a `'vers' (2)` resource identifies the version (and name) of a set of files which includes that file, thus linking all the files which make up the set. Apple uses this mechanism to identify System Software versions. All files on System Tools disks have a `'vers' (2)` resource that identifies the version of System Tools with which they were released. In addition, each file has a `'vers' (1)` resource that identifies the version of the particular file.

Version Resources and the Finder

The Finder displays the long message from `'vers' (1)` and `'vers' (2)` resources, if they are present, in the Get Info window of a file; it ignores the rest of the `'vers'` resource. Following is an example of the `'vers'` resources from Finder 6.1 with a Get Info window for the Finder file in Figure 1.

```
resource 'vers' (1) {
    0x06, 0x10, release, 0x00, verUS,
    "6.1",
    "6.1, Copyright Apple Computer, Inc. 1983-88"
};

resource 'vers' (2) {
    0x06, 0x03, release, 0x00, verUS,
    "6.0.3",
    "System Software Version 6.0.3"
};
```

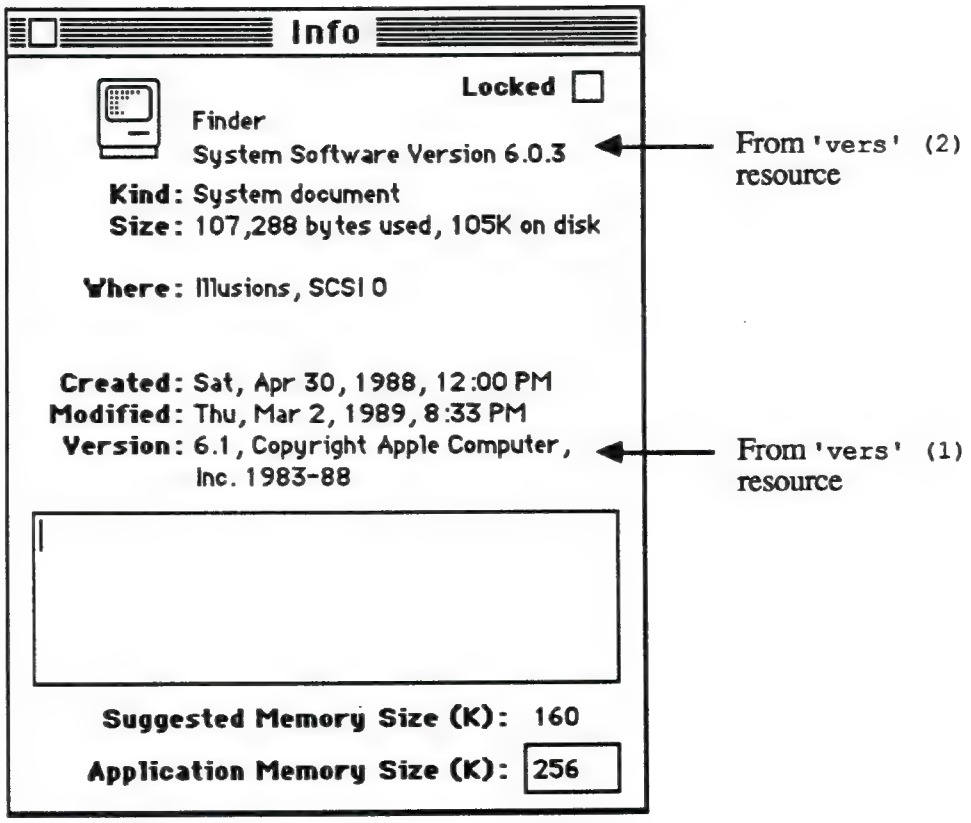


Figure 1–Get Info Window for the Finder File

The other fields (besides the long message) are often useful to applications other than the Finder. The short version number is good for displaying the version of a particular file, as the Finder does for the System and Finder in the About the Macintosh Finder window. The BCD version number is well suited for checking for a desired version number or comparing two versions. Note that this BCD numbering scheme represents a more recent version with a number greater than an older version, so a numeric comparison between two four-byte values is all that is necessary to determine which value is the most recent.

Final Note

The Finder Interface chapter of *Inside Macintosh*, Volume III-7 describes a resource (part of the bundle) that contains the **version data** of an application. This version data is typically a string that gives the name, version number, and date of the application. The Finder displays the version data (treating it as a string) in the Get Info window if there is no `'vers' (1)` resource in the application. Unlike this version data in an application, any type of file can contain `'vers'` resources, not just those files which contain bundles.

Further Reference:

- *Inside Macintosh*, Volume III-7, The Finder Interface
- Technical Note #48, Bundles

#190: Working Directories and MultiFinder

See also: The File Manager
Technical Note #77—HFS Ruminations

Written by: Darin Adler April 2, 1988

This technical note describes the way that working directories are handled under MultiFinder.

Some versions of Technical Note #77 claim that you can open working directories with a unique `ioWDProcID` and that they will only be deallocated when "the system is rebooted."

With MultiFinder, this has changed. When you call `PBOpenWD`, the `ioWDProcID` that you pass in is ignored. MultiFinder overrides your `ioWDProcID` with a unique process ID for your application, and deallocates all working directories that you allocated when your application terminates. Thus, you cannot use the `ioWDProcID` to identify your working directories when running under MultiFinder.

Indexing through working directories with `PBGetWDInfo` and a nonzero value of `ioWDProcID` is now a bad idea. This is because the working directories will have `ioWDProcID`s that MultiFinder has assigned, rather than the ones you specified.

Whenever you open a working directory with `PBOpenWD`, you should pass your application's signature as the `ioWDProcID` and close the working directory as soon as possible with `PBCloseWD`. You should only close working directories that you open, not ones that are returned to you from Standard File or `SysEnvirons`. It is best to keep working directories open for the minimum time necessary, and to avoid using them when possible. Note that working directories are implemented mostly for the benefit of old (pre-HFS) applications, and rarely need to be used.

#191: Font Names

See also: The Font Manager

Written by: Darin Adler April 2, 1988
Revised by: Bryan Stearns August 1, 1988

This note recommends the use of font names rather than font numbers.

The Font Manager chapter of *Inside Macintosh Volume IV* claims that font family numbers 0 through 127 are reserved for use by Apple, and numbers 128 through 255 are assigned by Apple for fonts created by software developers. This is no longer true. Developer Technical Support does **not** assign font family numbers. You should only use font numbers to reference the system font (font 0) and application default font (font 1). All other fonts should be identified by name. The Font/DA Mover will renumber a font when moving it into a file containing a conflicting font family.

The Font Manager routines `GetFontName` and `GetFNum` map font names to numbers and vice versa. This makes it simple to store a font's name in a document and turn it back into a number when reading the document. Unfortunately, `GetFNum` returns a 0 when a font by that name doesn't exist; this is the same as the font ID for the system font. The following routine in MPW Pascal alleviates this problem:

```
FUNCTION GetFontNumber(fontName: Str255; VAR fontNum: INTEGER) : BOOLEAN;
{GetFontNumber returns in fontNum the number for the font having
 the given fontName. If there's no such font, it returns FALSE.}

VAR
    systemFontName: Str255;

BEGIN
GetFNum(fontName, theNum);
IF fontNum = 0 THEN BEGIN
    {either the font was not found, or it is the system font}
    {if it was the system font, we got it, otherwise we didn't}
    GetFontName(0, systemFontName);
    GetFontNumber := EqualString(fontName, systemFontName, FALSE, FALSE);
END ELSE
    {if theNum was not 0, we found the font}
    GetFontNumber := TRUE;
END;
```

In MPW C:

```
Boolean GETFONTNUMBER(fontName, fontNum)
Str255* fontName;
short* fontNum;
```

```
/* GetFontNumber returns in fontNum the number for the font having
   the given fontName. If there's no such font, it returns false. */
{
    Str255* systemFontName;

    GETFNUM(fontName, fontNum);
    if (*fontNum == 0) {
        /* either the font was not found, or it is the system font */
        /* if it was the system font, we got it, otherwise we didn't */
        GETFONTNAME(0, systemFontName);
        return (EQUALSTRING(fontName, systemFontName, false, false));
    } else
        return true;
}
```

This routine makes it easy to find out if a given font exists; if the font isn't available, you can present a dialog to the user, allowing an appropriate substitute font to be chosen.

Handy Hint for Lists of Font Names

Most applications that offer the user a choice of fonts do so by creating a Fonts menu. Some applications, however, present a list of fonts in some other way: for example, word processors that use a dialog box to let the user pick a font family, point size, and style often display the font family choices in a List Manager list. You can get the Menu Manager to do most of the work by using `AddResMenu` to enumerate and alphabetize the names, as follows:

```
PROCEDURE BuildMyFontList;
CONST
     AnUnusedMenuID = 150; {a menu ID not used by any of your menus}
VAR
     tempMenu: MenuHandle;
     thisItem: INTEGER;
     aFontName: Str255;
BEGIN
     {Get a menu; use the Menu Manager to fill it with font names}
     tempMenu := NewMenu(AnUnusedMenuID,'x');
     AddResMenu(tempMenu,'FONT');

     {Extract the names we got, one at a time}
     FOR thisItem := 1 TO CountMItems(tempMenu) DO
          BEGIN
               {Extract the next name from the menu}
               GetItem(tempMenu,thisItem,aFontName);

               {** Do something with this font name (add   **}
               {** it to a List Manager list, or whatever) **}
          END;

     {We're done with the menu; dispose of it}
     DisposeMenu(tempMenu);
END; {BuildMyFontList}
```

This approach will help to insulate your application from changes to the Font Manager.

Historically, AddResMenu was modified to notice FOND resources at the same time the Font Manager began to support them, so applications that used this technique to build their font lists didn't need to be modified to work with FONDs.

Suggested Font Strategy for Applications and Documents

If your application offers the user a choice of a single font for use in the entire document, it's a simple matter to store the name of that font somewhere in the document (perhaps in an 'STR ' resource). However, if your application lets the user use many fonts within each document (as is the case with most word processors) a more complex strategy is necessary: the Font Name Mapping Table. Each entry might look like this:

```
FontMapEntry : RECORD
    name: Str255;       {The name of the font}
    localID: INTEGER;   {a unique number for this entry}
    realID: INTEGER;    {last time we checked, this font's font number}
    useCount: INTEGER;  {How many times this font is used in this document}
END;
```

In a new document, start out with no entries in the table. When the user changes a selection of text to a new font (that is, one whose name is not in the table), add an entry to the table. Set useCount to 1 (as this font is now referenced once within the document), and pick a localID that is unique within the table. In the text, or wherever you would normally keep the font number, store a copy of this localID instead of the font number. Use GetFontNumber (the example above) to get the "real" font number, and store it in the table as well, in realID.

Whenever you need to draw text, search through the table for the proper localID. When you find it, use the realID that is stored in that entry in a call to TextFont, then draw your text as usual.

Keep the useCount updated, so that you know when to get rid of a table entry. If the user deletes a range of text, or changes it to another font, examine the text to see if you should decrement any of the useCounts in your table. When a useCount for an entry becomes zero, you'll know that the font for that entry isn't used anywhere within the document, and you can remove the entry from the table.

When you save the document, save the table with it. When you open an existing document, load the table. For each entry, call GetFontNumber using each name, and update the realID field with the current font number for that name. If a font isn't present, you should warn the user: You could let the user choose an alternative font, or use the default application font by storing (and using) the constant applFont as that font's realID; this way, the user could still edit the document, but the original font name would remain, so that when the user adds the proper font to the System file, or moves the document to a Macintosh whose System file contains it, the document would be displayed as originally intended.

The overhead of handling your documents' fonts in this manner is rather small; as more font families become available, and Font/DA Mover's renumberings occur more often, your customers will appreciate this extra effort.

Macintosh
Technical Notes



Developer Technical Support

#192: Surprises in LaserWriter 5.0 and Newer

Revised by: Mary Burke & Scott "Zz" Zimmerman February 1990
Written by: Scott "Zz" Zimmerman April 1988

This Technical Note describes some changes in version 5.0 and later LaserWriter drivers.
Changes since April 1988: Described a bug in 5.x which is fixed in 6.0 and later, and reiterated a warning about storing fonts in an application.

With the release of LaserWriter 5.0 and background printing, a few changes had to be made to the LaserWriter driver. Although these changes were transparent to most applications, some have had problems. Most of these problems are related to use of unsupported features. This Note details a partial list of the changes.

No Mo' Low

Because of the problems supporting both the high-level and low-level interfaces in the background, the low-level interface is all but removed. Instead of the low-level calls being executed by the device driver, the `_PrCtlCall` procedure converts the call into its high-level equivalent before execution. This way, the LaserWriter driver has a common entry point for both the low-level and high-level interfaces. Because of this conversion, the low-level calls may not be faster than using the high-level equivalents. In some cases, they may even be slower.

Version 5.x of the LaserWriter driver also contains a bug with the low-level interface. If an application which uses the low-level Printing Manager interface encounters an error during the course of the print job, the LaserWriter driver crashes before the application has a chance to see the error. Because the error occurs inside the driver, there is no way for an application to predict or work around this problem. The only solution to this problem is to use the high-level Printing Manager interface or to upgrade to version 6.0 or later of the LaserWriter driver which fixes this bug.

Are You Convertible?

Whereas the conversion of the low-level calls should be transparent, the conversion routines make some assumptions. The conversion routines require a context in which to operate; the Printing Manager maintains a certain state while executing commands, and the conversion routines need access to this state to perform the conversion. To provide this context, an application must have opened a document and a page. This requirement means that the original method of using the low-level interface, which is documented in *Inside Macintosh*, Volume II-164, no longer works, as in the following example:

```
PrDrvrOpen;
PrCtlCall(iPrDevCtl, lPrReset, 0, 0);
{ Send data to be printed. }
PrCtlCall(iPrDevCtl, lPrPageEnd, 0, 0);
PrDrvrClose;
```

Instead, an application should use the following:

```
PrDrvrOpen;
PrCtlCall(iPrDevCtl, lPrDocOpen, 0, 0);
PrCtlCall(iPrDevCtl, lPrPageOpen, 0, 0);
{ Send data to be printed. }
PrCtlCall(iPrDevCtl, lPrPageClose, 0, 0);
PrCtlCall(iPrDevCtl, lPrDocClose, 0, 0);
PrDrvrClose;
```

This method provides the Printing Manager with the context it needs to convert the calls.

Really Unsupported Features

Sending data to the printer between the `_PrOpenDoc` or `lPrDocOpen` and the `_PrOpenPage` or `lPrPageOpen` calls is not currently, and has never been supported. LaserWriter drivers prior to 5.0 interpreted this data, but 5.0 and later drivers ignore it. To download an application-specific PostScript® dictionary as a header with each document, Apple recommends that the application provide a `'PREC'` resource of `ID = 103`, as is described in the *LaserWriter Reference*.

A Little Less Control

Four of the six printer control calls originally supported by the LaserWriter driver have been discontinued due to lack of use and difficulty supporting with background printing. The four calls which follow were only supported by the LaserWriter driver and only documented in the *LaserWriter Reference Manual*:

`°•° fill` `°•° hexBuf` `°•° printR` `°•° printF`

In addition to these calls, the `stdBuf` call is also affected. There are two versions of the `stdBuf` call depending upon the sign of the `bytes` parameter. If `bytes` is negative, the text passed to the `stdBuf` call is converted to PostScript text before being sent to the LaserWriter. This conversion means that special PostScript characters in the text are preceded by a PostScript escape character. In addition, characters with an ASCII value greater than 128 are converted to octal before being sent to the LaserWriter. This version of the call is no longer supported.

If the bytes parameter is positive, the text passed to the call is sent directly to the LaserWriter without conversion and interpreted as PostScript instructions. This version of the call is still supported, but there is one more problem. When an application first opens the low-level driver (via `_PrDrvrOpen`) with background printing enabled, no clip region is defined. If the application then begins sending PostScript to the driver via the `stdBuf` call, all of the output is clipped, and only a blank page is printed.

To prevent this problem, the application must force a clip region to be sent to the LaserWriter. The region is sent by the driver when it receives its first drawing command. Unfortunately, the driver does not consider the `stdBuf` call to be a drawing command. To force the clip region on the printer, the application can use the `iPrBitsCtl` call to print a small bitmap outside the printable area of the page. This call does not have any effect on the document, but it fires the bottleneck

routine and causes a definition of the clip region. Since the clip region is reinitialized at each call to `lPrPageOpen`, the application should send the bitmap once at the start of each page. If any other printer control calls precede the `stdBuf` call, the application does not need to send the bitmap.

Background Preparations

The `'PREC' ID = 201` mechanism only works when background printing is disabled. This limitation is because of difficulties finding the resource under MultiFinder. Since the option only works in the foreground, and since there is no way for an application to know if background printing is enabled, an application should avoid using this feature.

Fonts In An Application?

There are two problems when printing application fonts with the LaserWriter driver. *Application fonts* are fonts that are stored in the resource fork of the application's resource file rather than being stored in the System file. The first problem occurs when the application font has the same name as the application's resource file. If this is the true, the LaserWriter driver incorrectly assumes that the application's resource file is a font file, and closes it after using the font. To solve this problem, developers should make sure the name of the application font (i.e., the `'FOND'` resource) is different from the name of the application's resource file. Since the application can still be renamed by the user, developers should try to select a unique font name. If the font name does not appear in a font menu, developers can simply append the application's creator string onto the desired font name (i.e., `'MyApplicationFont ZZAP'`).

The second problem with application fonts only occurs when background printing is enabled. When a print job is performed in the background, the LaserWriter driver writes all of the data to be printed into a file (called a spool file) for printing at a later time by Print Monitor. Since the LaserWriter driver has no way of knowing when the file will actually be printed, it cannot assume that the application will still be open when the job is printed. This is a problem if the application contains application fonts that Print Monitor needs at print time. To solve this problem, the LaserWriter driver must determine whether the fonts needed by the application are resident in the System file or in the application file. If they are in the application file, the driver must copy them into the spool file so they are available to Print Monitor. This practice can lead to very large spool files, as well as a significant loss of performance when background printing is enabled. To solve these and other user interface problems related to application fonts, Apple strongly recommends that developers ship custom application fonts as suitcase files for the user to install in the System file.

Headin' For Trouble

There is a minor bug in version 5.0 of the LaserWriter driver that only affects applications that parse the PostScript header downloaded by the driver with each document. This header contains some PostScript comments that provide information about the current job. One of these comments is `IncludeProcSet`. This comment takes three arguments: a PostScript dictionary name, a major version number, and a minor version number. In version 4.0 of the LaserWriter driver, the comment line looked like the following:

```
%% IncludeProcSet: (Appledict md) 65 0
```

Unfortunately, in version 5.0 of the LaserWriter driver, the last argument was removed. This caused the comment line to look like the following:

```
%% IncludeProcSet (Appledict md) 66
```

Since Adobe defined the comment to take three arguments, it is reasonable for applications that parse the comments to expect three arguments; therefore, version 5.1 and later of the LaserWriter driver contain the correct version of the comment:

```
%% IncludeProcSet (Appledict md) 67 0
```

No Go With Zero

Some applications want to force a font to be downloaded to the LaserWriter without actually printing characters with the font. This can be done in three easy steps:

1. Save the current pen position.
2. Use any text drawing routine to draw a space character.
3. Move the pen back to the saved position.

Some applications use `_DrawString` with a empty string (e.g., `DrawString('')`) to force the font downloading. Although this worked in LaserWriter drivers up to 5.0, these calls are ignored by the 5.1 and later drivers. The main reasons for this change were optimization of performance and a reduction in the size of spool files.

Further Reference:

- *Inside Macintosh*, Volumes II & V, The Printing Manager
- *LaserWriter Reference Manual*

PostScript is a registered trademark of Adobe Systems Incorporated.

Macintosh
Technical Notes

Developer Technical Support

#193: So Many Bitmaps, So Little Time

Revised by: Rich Collyer
Written by: Rick Blair

December 1989
April 1988

This Technical Note discusses the routine `BitMapToRegion`, which converts a bitmap to a region, and is available in the 32-Bit QuickDraw INIT and from Apple Software Licensing.
Changes since October 1989: Added trap definitions for developers using the 32-Bit QuickDraw version of this routine without the correct MPW include file.

The following routine is now available to convert a bitmap to a region:

```
FUNCTION BitMapToRegion(region:RgnHandle; bMap:BitMap): OSErr;
```

in C:

```
pascal OSErr BitMapToRegion(RgnHandle region, BitMap bMap);
```

If you are using the 32-Bit QuickDraw version of this routine without the correct MPW include file, then you need to include one of the following definitions:

Pascal

```
FUNCTION BitMapToRegion (region: RgnHandle; bMap: BitMap): OSErr;
         INLINE $A8D7;
```

C

```
pascal OSErr BitMapToRegion (RgnHandle region, const BitMap *bMap)
            = 0xA8D7;
```

Assembly

```
_BitMapToRegion         OPWORD         $A8D7
```

The region will be built so that all one bits in `bMap` are inside the region and all zero bits are outside of it.

As with all QuickDraw calls which change a region, `BitMapToRegion` requires you to pass an existing region (originally created by `_NewRgn`). If the region cannot be built due to an insufficient heap space or a size greater than 32K, then the routine will return an appropriate error code and the region will be empty. If the region would have exceeded 32K, the error will be `rgnTooBigErr` (-500).

This function is useful for a number of situations where you have (or can produce) a bitmap representing an area. You can use `_CalcMask` to produce such a bitmap. Once you have a

region, you can perform region operations (i.e., _PtInRgn, _UnionRgn, or _InsetRgn) or call _DragGrayRgn, for example.

This call is part of the 32-Bit QuickDraw INIT ($A8D7). If you do not wish to depend on 32-Bit QuickDraw, then you can obtain a version of BitMapToRegion in MPW object format which can be linked into an MPW program, by contacting Apple Software Licensing:

Apple Software Licensing
Apple Computer, Inc.,
20525 Mariani Avenue, M/S 38-I
Cupertino, CA, 95014
(408) 974-4667
AppleLink: SW.LICENSE

If you licensed the older version of this routine, BitMapRgn, contact Software Licensing about receiving an updated version. We recommend you update your application to use the new version as soon as possible.

The new version is now named BitMapToRegion to be consistent with the version in 32-Bit QuickDraw and the MPW interfaces. In addition, BitMapToRegion offers new features. You can now pass a one-bit pixelmap which has been coerced to a bitmap. If you pass a pixelmap which is too large, then you will get a pixmapTooDeepErr (-148) error. You can also pass the portBits of a window, much like you would do with a call to _CopyBits.

There is a potential problem with this routine, since MPW 3.1 include files contain information about 32-Bit QuickDraw. If you want BitMapToRegion to be available on all machines, then you must use the object file from Software Licensing. The problem is that when you compile your application with MPW 3.1 or later, the 32-Bit QuickDraw version gets preference over the object file. You **must** comment out the routine in the include files if you want to use the object file. If you only care about using BitMapToRegion on machines running 32-Bit QuickDraw, then you need not do anything.

Macintosh Technical Notes

#194: WMgrPortability

See also: The Window Manager
QuickDraw
MultiFinder Development Package

Written by: Rick Blair April 2, 1988

Where `WMgrPort` (the Window Manager's port), MultiFinder, and drawing outside of one's windows will be reconciled.

Beware

Drawing outside of windows from within an application is guaranteed to make that application less compatible with future systems. In order to be as MultiFinder compatible as possible, draw only in response to an update event or as part of the feedback for a user action, i.e. while tracking the mouse. MultiFinder compatibility is just as important as HFS compatibility!

MultiFinder documentation warns against drawing in `WMgrPort` since the system "owns" the desktop and windows of other applications besides your own are drawn within it. This note will tell you how and when to draw outside the confines of your own windows if you feel that you must.

In the future the system may provide calls for drawing outside your windows safely. When that occurs, the techniques described here may no longer be valid. Nevertheless...

WMgrPort and GrayRgn

`WMgrPort` has its `visRgn` set to include all active screens. Its `clipRgn` is initially set to "wide open" (the rectangle –32767, –32767, 32767, 32767), although Window Manager routines like `ClipAbove`, etc. will change it. Consider this `GrafPort` read-only. The global variable `GrayRgn` is a region which is equal to the `WMgrPort`'s `visRgn` minus the menu bar area.

Note that you should use `GrayRgn`, which is the best way to find out the shape, size, and coordinates of the screens. You will never have to use the `WMgrPort` directly, and should not call `GetWMgrPort` under any circumstances.

Rules

Only draw to the whole screen/desktop in a "modal" way. This can take the form of a brief animation across windows or the visual feedback for dragging from one window to another. It is important to know that no other application (including the Finder) will draw until you have finished. To guarantee this, you must follow some rules:

In the case of a drag, you should only draw while the mouse button is down. In the case of an animation effect, the drawing should be of brief duration. All operations should conclude with nothing left drawn outside your windows. Under MultiFinder (version 1.0 and 6.0 at least) you will be OK if you don't call `GetNextEvent`, `EventAvail`, or `WaitNextEvent` while drawing outside your windows. Use the `StillDown` function (or the `WaitMouseUp` function) for loops that wait for the mouse button to go up. Remember, however, it is only through possible future system-provided calls that you can be completely safe from others drawing underneath you.

Never draw something on the desktop and leave it there. There is no way to tell the system that you have drawn on that bit of desktop, so the Finder will draw right over you.

Examples

The most famous animation effect is the `ZoomRect` routine. It is used by the Finder to draw a series of nested rectangles around an icon that is being opened. The rectangles form a progression (zoom) out to where the window for the icon will be placed.

Another, potentially more interesting, case is where you want to drag something from one window to another, perhaps to copy it. This is often done with `DragGrayRgn`, which for this purpose will do the right thing (not call `GetNextEvent`, etc.).

How to do these effects

Use a `GrafPort` (not a window or the `WMgrPort`) that covers all the screens. `OpenPort` will set up most of the fields of the `GrafPort` properly. All you have to do is change the `visRgn` of your port to a copy of `GrayRgn` and put the `GrayRgn`'s `rgnBBox` into your `portRect`. Directly manipulating the `visRgn` of a **window** is a no-no under MultiFinder.

Draw using `srcXor` mode. This will allow you to erase as you go, by drawing each object a second time, also in `srcXor` mode. You must leave all areas outside your windows exactly as you found them.

WDEFs and MDEFs

Window and menu definition procedures draw in the current port, which is set to the `WMgrPort` by the Window Manager and the Menu Manager. Note that this means that you do not ever have to call `GetWMgrPort`, as mentioned above. We recommend that you never draw into it except from one of these procedures.

#195: ASP and AFP Description Discrepancies

See also: The AppleTalk Manager

Written by: Mark Bennett August 1, 1988

The descriptions of the AppleTalk Session Protocol and AppleTalk Filing Protocol functions within the body of the AppleTalk Manager chapter are incorrect and conflict with those in the Summary of the AppleTalk Manager. This technical note resolves the discrepancy.

The descriptions of the AppleTalk Session Protocol and AppleTalk Filing Protocol functions which are described on pages 534 through 548 of *Inside Macintosh Volume V* conflict with the descriptions in the Summary of the AppleTalk Manager section, pages 554 through 559. The descriptions in the Summary of the AppleTalk Manager section are correct and should be followed.

The Summary of the AppleTalk Manager does not, however, present a description of the correct meaning of the arrows next to the parameter names in the function descriptions. The meaning of the arrows is equivalent to the one given to them in the descriptions of other Operating System calls, i.e.:

Arrow	Meaning
→	Parameter is passed
←	Parameter is returned
↔	Parameter is passed and returned

Macintosh
Technical Notes

Developer Technical Support

#196: 'CDEF' Parameters and Bugs

Revised by: David Shayer — October 1989
Written by: Mark Bennett — August 1988

This Technical Note describes known bugs in the Control Manager which affect control definition functions (`'CDEF'` resources).
Changes since August 1988: Updated to reflect known bugs in the `posCntl` and `thumbCntl` messages and the Control Manager `_TrackControl` call.

The Control Manager chapter of *Inside Macintosh*, Volume I-309, describes how to write a control definition function (`'CDEF'` resource). This Note assumes a basic understanding of this chapter, specifically of the various messages which are sent in the `message` parameter.

drawCntl (0) and autoTrack (8)

When a `'CDEF'` is called with either the message `drawCntl` or `autoTrack`, it is possible for the high word of the `param` parameter to contain undefined data which could result in the failure of routines that rely upon all 32 bits of `param` being defined. `'CDEF'` resources should only consider the low word of the `param` parameter when dealing with the `drawCntl` and `autoTrack` messages.

posCntl (5) and thumbCntl (6)

According to *Inside Macintosh*, the Control Manager calls a `'CDEF'` with the `posCntl` message and the `thumbCntl` message if an application does custom dragging of an indicator (a thumb), but not if it does default dragging. This is not true. The Control Manager calls a `'CDEF'` with the `posCntl` message if an application does default dragging, which is exactly the opposite of the way it is documented. The `'CDEF'` receives the `thumbCntl` message regardless of which type of dragging an application does, however, the results are used only for default dragging (they are ignored for custom dragging).

_TrackControl

When a user clicks on your control, you normally call `_TrackControl`, which is supposed to return zero if the user does not change the control's setting or the part code if the user does change the setting. For `'CDEF'` resources that implement custom dragging, `_TrackControl` returns zero whether or not the user changes the control's setting. To work around this problem, you must use another method to find out if the user has changed the control's setting, such as comparing the control's value before and after the call to `_TrackControl`.

Further Reference:

- *Inside Macintosh*, Volume I-309, The Control Manager

#197: Chooser Enhancements

See also: The Device Manager (Volumes II, IV, and V)
The Control Panel (Volume V)

Written by: Chris Knepper August 1, 1988

Beginning with version 3.2, the Chooser has been enhanced to provide support for additional controls.

As stated in *Inside Macintosh* IV-217, the Chooser communicates with device packages as if they were the following function:

```
FUNCTION Device (message, caller: INTEGER;
                 objName, zoneName: StringPtr;
                 p1, p2: LONGINT): OSErr;
```

This function is contained in the device package's 'PACK' -4096 resource. If bit 17 in the `flags` field of this 'PACK' is set, the `Device` function will receive an initialization message when the user selects that device resource file from the Chooser's window – `Device` will be called with `message = initMsg`. This is the first message that device packages receive and may be used to set up default configurations. The MPW assembler interface for this new message is:

```
initMsg EQU 11
```

When the `Device` function receives `initMsg`, the `objName` parameter contains a pointer to an array of `ControlHandle`s. For this message (and for the `buttonMsg` described below) the `objName` parameter is a pointer to a structure as follows: it begins with a size word and is followed by at least 4 `ControlHandle`s. The size is at least 18 bytes (2 bytes for the size word and 4 bytes each for the handles). More handles may be added in the future. The four `ControlHandle`s that have been defined thus far are the left and right buttons and the "on" and "off" radio buttons. Their handles appear in this order:

size word
left `ControlHandle`
right `ControlHandle`
on `ControlHandle`
off `ControlHandle`
...?

The `Flags` bits of the 'PACK' control which buttons are used. Bits 26 and 27 are used to indicate whether you use the left and right buttons and are described in the Device Manager chapters in *Inside Macintosh* volumes IV and V.

Bit 20 tells the Chooser that you are employing the on and off radio buttons. Their titles can be something other than "on" and "off", of course, but we'll continue to use those names since that is what the LaserWriter driver (5.0 or later) calls them. In addition to the controls and their titles, a static label ("Background Printing:" for the LaserWriter) will be displayed.

The title strings for these radio buttons are contained in the 'STR ' resources with IDs –4089 and –4088 in the device resource file. The string for the label is contained in 'STR ' resource –4087.

The rectangles for these items are defined in the 'nrct' –4096 resource in the device resource file. The Device Manager chapter of *Inside Macintosh* V-430 describes the 'nrct' resources. The third and fourth rectangles position the radio buttons while the fifth rectangle positions the label.

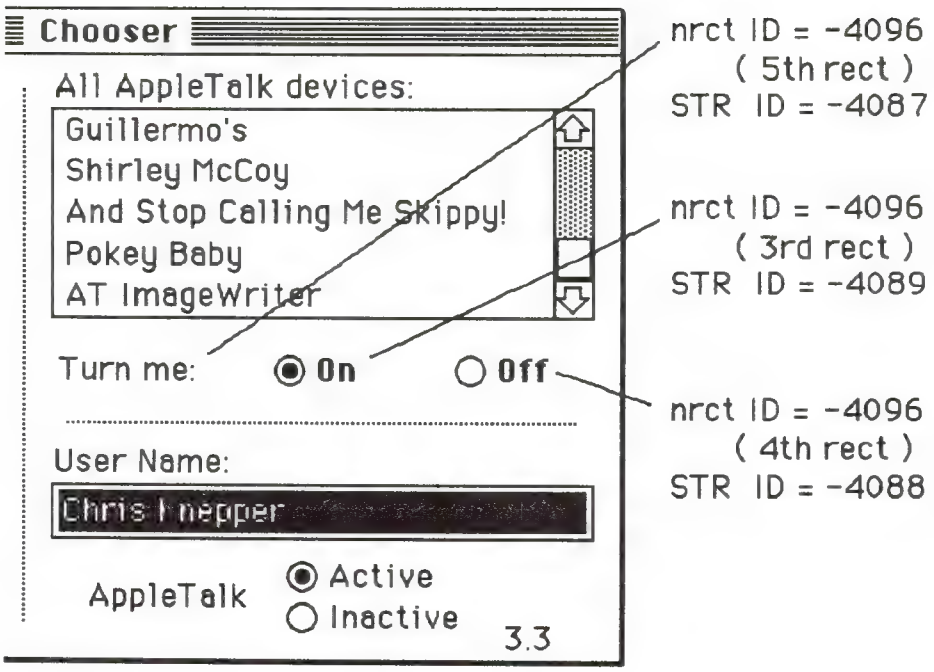
For example, the Chooser interface pictured below corresponds to the following Rez input.

```
/* label for radio buttons */
resource 'STR ' (-4087) {
            "Turn me:"
};

/* "off" radio button */
resource 'STR ' (-4088) {
            "Off"
};

/* "on" radio button */
resource 'STR ' (-4089) {
            "On"
};

resource 'nrct' (-4096) {
            {
            /* [1] Left Button */
            {0, 0, 0, 0},
            /* [2] Right Button */
            {0, 0, 0, 0},
            /* [3] on Radio Button */
            {114, 260, 130, 320},
            /* [4] off Radio Button */
            {114, 330, 130, 390},
            /* [5] Label for Radio Buttons */
            {114, 177, 130, 250}
            }
};
```



If the rectangles for these items are not specified in the 'nrct' resource, then the Chooser uses default positions for these items from the Chooser's private 'nrct' resource. These default positions are:

```
/* [3] on Radio Button */
{114, 272, 130, 332},
/* [4] off Radio Button */
{114, 335, 130, 395},
/* [5] Label for Radio Buttons */
{114, 170, 130, 267},
```

In order for the radio buttons to be displayed, the 'PACK' must call `ShowControl` for each of them, and pass in the corresponding `ControlHandle`. To turn one of the radio buttons "on" and the other "off," the 'PACK' must call `SetCtlValue`. The best time to call these routines to set up the radio buttons is when the 'PACK' receives the `initMsg`. By showing the buttons after you have set their values, you can avoid unnecessary redrawing.

When the user clicks either of the radio buttons, the 'PACK' receives `buttonMsg` and `objName` contains a pointer to an array of `ControlHandles`, as explained above. The 'PACK' knows which of the radio buttons was clicked through the low-order byte of the `p2` parameter. This byte contains 3 if the "on" radio button was clicked, or 4 if the "off" radio button was clicked. It is the device package's responsibility to set or clear the radio buttons in response to the user's action.



#198: Font/DA Mover, Styled Fonts, and 'NFNT's

Updated by: Joseph Maurer
Written by: Bryan Stearns

May 1992
August 1988

This Note formerly discussed issues concerning the arrangement of font-related resources and their IDs. The subject has been updated and included in the new Technical Note #26: "Fond of FONDs".

#199: KillNBP Clarification

See also: The AppleTalk Manager

Written by: Mark Bennett August 1, 1988

This technical note clears up some confusion regarding the Name Binding Protocol KillNBP function.

The description of the `PKillNBP` function on page 519 of *Inside Macintosh Volume V* is somewhat confusing. The data type of the parameter `thePBptr` is incorrectly given as `ATPPBPtr` and the pointer to the queue element from the NBP call to be aborted is incorrectly given as being passed in `aKillQEl`. The following is a correct description of the `KillNBP` function:

KillNBP function

```
FUNCTION PKillNBP (thePBptr: MPPPBPtr; async: BOOLEAN) : OSErr;
```

Parameter block

→	26	`csCode`	word	Always `PKillNBP`
→	28	`nKillQEl`	pointer	Pointer to queue element



#200: MPW 2.0.2 Bugs

Written by: Dave Burnard — August 1, 1988
Modified by: Andy Shebanow — October 1, 1988

This Technical Note describes latest information about bugs or unexpected "features" in the MPW C, Pascal, and Assembler products and the Toolbox and OS Interface Libraries. We intend this Note to be a complete list of all known bugs in these products, which will be updated as old bugs are fixed, or new ones appear. If you have encountered a bug or unexpected feature which is not described here, be sure to let us know. Specific code examples are useful.

The bugs described in the October 1 revision of this Note will be fixed in the 3.0 release of MPW scheduled for Fall 1988.

Changes since August 1, 1988: Corrected the description of "bug" #3 under MPW C as it is not a bug according to the definition of the C language and corrected an error in bug #2 of the Interface Libraries concerning the glue for `_SlotVInstall` and `_SlotVRemove`.

C Language

The following information applies to the C compiler and its associated libraries shipped with the 2.0.2 version of MPW.

1) A series of bugs involving floating point array elements and the +=, *=, and = operators. A similar bug was reported as fixed in MPW 2.0.2, unfortunately the fix did not apply to array elements. This bug ONLY occurs when using SANE in combination with `float` or `double` variables, it does not occur if the `-mc68881` compiler option is specified or if `extended` variables are used. The following fragment illustrates the bugs:

```
main()
{
        double x[2],y;  /* Also fails if x,y are declared float
                           but succeeds if declared extended */
        x[0] = 0.5;
        y = 5.0;
        x[0] += 2.0*y;
        printf("x[0] = %f\n", x[0]);

        x[0] = 0.5;
        y = 5.0;
        x[0] = x[0] + 2.0*y;
        printf("x[0] = %f\n", x[0]);
```

```
        x[0] = 0.5;
        y = 5.0;
        x[0] *= 2.0*y;
        printf("x[0] = %f\n", x[0]);

        x[0] = 0.5;
        y = 5.0;
        x[0] = x[0]*(2.0*y);  /* Succeeds if parenthesis are removed */
        printf("x[0] = %f\n", x[0]);

        exit(0);
}
```

This code fragment returns the erroneous values 0.5, 0.5, 0.5, 0.5 (the correct values are 10.5,10.5, 5.0 and 5.0).

Workaround: If using SANE, use `extended` variables in these situations.

2) Taking the address of a floating point formal parameter (function argument) to a function fails. This bug occurs when using either SANE or the `-mc68881` compiler option in combination with `float` or `double` function arguments, it does not occur if the function arguments are declared `extended`. The following fragment illustrates two instances of this bug:

```
#include <Types.h>
#include <Math.h>
#include <stdio.h>

#define real float    /* Fails with either float or double */

main()
{
        Bug1(1.0, 2.0, 3.0);
        Bug2(1.0, 2.0, 3.0);
}

Bug1 (x, y, z)
        real x, y, z;
{
        real *p, *q, *r;

        /* Take address of arguments, assign directly */
        p = &x; q = &y; r = &z;

        fprintf(stderr, "Example 1: Before: %g, %g, %g\n", x, y, z);
        *p = 11.0;
        *q = 12.0;
        *r = 13.0;
        fprintf(stderr, "Example 1: After:  %g, %g, %g\n", x, y, z);
}

Bug2(x, y, z)
        real x, y, z;
{
        fprintf(stderr, "Example 2: Bug2 Before: %g, %g, %g\n", x, y, z);

        /* Take address of arguments, assign indirectly */
        foo(&x, &y, &z);
        fprintf(stderr, "Example 2: Bug2 After:  %g, %g, %g\n", x, y, z);
}
```

```
foo(x, y, z)
        real *x, *y, *z;
{
        fprintf(stderr, "Example 2: foo Before: %g, %g, %g\n", *x, *y, *z);
        *x = 11.0;
        *y = 12.0;
        *z = 13.0;
        fprintf(stderr, "Example 2: foo After:  %g, %g, %g\n", *x, *y, *z);
}
```

This is, in fact, a general problem with C compilers. The underlying reason for this problem is related to the automatic widening and narrowing of basic types performed by C compilers. For instance in C, as defined by K&R (*The C Programming Language*, Kernighan & Ritchie, 1978, Appendix A Sec. 7.1, p186 and Sec. 10.1, p206), variables of type `char` and `short` are widened to `int` before being passed as function arguments and narrowed before use inside the function. Similarly, the floating point type `float` is automatically widened to `double` before being passed. K&R notes, however, that "C converts all `float` actual parameters to `double`, so formal parameters declared `float` have their declarations adjusted to read `double`." The value of such a formal parameter is **not** narrowed to the declared type of the parameter, instead the declared type is **adjusted** to match that of the widened value. So, in fact, the sample code above will fail if `real` is defined as `float`, even on a bug free K&R conforming compiler.

In MPW C, where `float` and `double` are widened to `extended`, the sample code fails for either `float` or `double` formal parameters. This can, of course, lead to additional problems if you are porting code from an environment where `double` was the widened floating point type (where taking the address of a double formal parameter would work as expected).

Workaround: Taking the address of a function argument is not recommended; you should make a local copy in a temporary variable and take the address of the copy. If you must take the address of a floating point function argument, make sure it is declared as type `extended`, and the pointer is of type `extended*`.

3) The shift operators >> and << can sometimes produce unexpected results when applied to unsigned short and unsigned char operands. The anomaly lies in the fact that the 680x0 `LSR.L` and `LSL.L` instructions are used instead of the `LSR.W` and `LSL.W` or the `LSR.B` and `LSL.B` instructions. The following example illustrates this anomaly:

```
main()
{
        unsigned short u, c;
        short i, k;

        u = 0xFFFF;
        k = 8;

        i = (u << k)/256;
        printf("unsigned short: i = %d, %#x\n", i, i);

        c = 256;
        i = (u << k)/c;
        printf("unsigned short: i = %d, %#x\n", i, i);
}
```

This code fragment returns the values -1, 0xFFFFFFFF and -1, 0xFFFFFFFF, which are the correct values as defined by the C language, however, you might be expecting 255, 0xFF and 255, 0xFF.

4) The compiler optimization flags `-q` (sacrifice code size for speed) and `-q2` (memory locations won't change except by explicit stores) can produce incorrect code. We have several vague descriptions of problems, and are looking for more specific examples. The following example illustrates a specific bug that occurs when using enumerated types:

```
badfunc(bool, i)
        Boolean *bool;
        int i;
{
        while (true) {
                if(i < 3) {
                        *bool  = true;
                        if (func(1)) return;
                }
                if (func(i)) return;
        }
}
```

The enumerated type here is the type used to define the values `true` and `false` in the header file `Types.h`. The optimizer is apparently confused by the fact that `true` has the `char` value `1`, which it thinks is the same as the `int` value `1` to be passed to the function func(). The object code produced for the two calls to the function `func()` is:

```
...
MOVEQ       #$01,D3           ; Move the constant 1 into a register
...
MOVE.B      D3,(A2)           ; Assign "true" to variable bool
MOVE.B      D3,-(A7)          ; Attempt to push integer 1 onto stack
                              ; ERROR should be MOVE.L !!!!!
JSR         *+$0002           ; JSR to func
...
MOVE.L      D4,-(A7)          ; Correctly push integer variable i onto stack
JSR         *+$0002           ; JSR to func
...
```

In the first function call, the `int` constant `1` is passed as a byte value! Since the stack is correctly adjusted after the first call this error may go undetected (except that the called function may spot the resulting nonsensical parameter). This problem is, of course, not limited to the enumerated type defining `true` and `false`, but can occur as a side effect of any of the many enumerated types defined in the Toolbox header files.

Workaround: The best solution, for now, is to avoid using the optimization flags `-q` and `-q2` altogether.

5) The compiler flag `-mc68020` (generate MC68020 instructions) generates inconsistent code when passing structures by value. Specifically, structures larger than 4 bytes that are not a multiple of 4 bytes are padded out to the nearest 4 byte multiple before being pushed onto the stack by the calling routine. Unfortunately, the called routine does not take this padding into account when

accessing its function arguments. The following example illustrates this bug:

```
#include <Types.h>
#include <strings.h>

typedef struct          /* 6 byte long structure */
{
  short Temp1;
  short Temp2;
  short Temp3;
} TestStruct;

main()
{
  TestStruct reply;

  foo(reply,"Hello world.\n");
}


foo(tst ,str)
  TestStruct tst;
  char *str;
{
  Debugger(); /* So we can look, before stepping off the cliff */
  printf("%s",str);
}
```

Since function arguments are pushed onto the stack in left to right order in C, the pointer to the string constant `"Hello world.\n"` is pushed onto the stack before the padded contents of the `reply` structure. Thus when the called function, `foo()`, goes to fetch the argument `str`, it looks at the location just beyond where the argument `tst` is located. Unfortunately, since the called function does not know the structure argument was padded, it will not find the correct value for the second argument (or in the general case, any arguments following the structure).

Workaround: When using the `-mc68020` compiler option, either don't pass structures larger than 4 bytes by value, or make sure all structures larger than 4 bytes are padded out to a multiple of 4 bytes.

6) Switch statements with non-integer controlling expressions can fail. The following example illustrates the problem:

```
#include <stdio.h>
#include <strings.h>
main()
{
        unsigned short w = 1;

        printf("The following test fails in MPW\n");
        printf("Should switch to 1; actually got to ");
```

```
        switch (w) {
                case 1:
                        printf("1\n");
                        break;
                case 5:
                        printf("5\n");
                        break;
                case 32771:
                        printf("32771\n");
                        break;
                case 32773:
                        printf("32773\n");
                        break;
                default:
                        printf("default\n");
                        break;
        }
}
```

In this example, instead of reaching `case 1:` in the switch statement, the `default:` case is reached due to the fact that the compiler generates comparison instructions based on word length values. The 1st Edition of K&R (*The C Programming Language*, Kernighan & Ritchie, 1978, Appendix A Sec. 9.7, p202) requires all case constants to have integer type and the controlling expression to be coerced to integer type. The 2nd Edition (ANSI) of K&R (*The C Programming Language*, Kernighan & Ritchie, 1988, Appendix A Sec. 9.4, p223) requires that the controlling expression and all case constants undergo "integral promotion"—promotion to `int` (or to `unsigned int` if necessary). MPW 2.0.2 C fails to promote either the controlling expression or the case constants to type `int`.

Workaround: If the controlling expression is manually coerced to type `int` this example functions correctly.

7) Variable declarations inside the body of switch statements, when combined with the `-g` compiler flag, can cause the compiler's code generator to abort with the message: "Code generator abort code 615." The following example illustrates the problem:

```
foo(i)
        int i;
{
        switch (i) {
                int j;   /* VARIABLE DECLARATION INSIDE BODY OF SWITCH */


                case 0:
                        j = 22 +i;
                        printf("INSIDE: i=%d, j=%d\n", i, j);
                        break;
                case 1:
                        j = 57 -i;
                        printf("INSIDE: i=%d, j=%d\n", i, j);
                        break;
                default:
                        j = i;
                        printf("INSIDE: i=%d, j=%d\n", i, j);
                        break;
        }
}
```

While such a declaration is perfectly legal C, it is a bad practice (*The C Programming Language*, Kernighan & Ritchie, 1978, Appendix A Sec. 9.2, p201). K&R go on to point out that if the declaration of the variable contains an initializer, the initialization will NOT take place. This is also the ANSI draft C standard's interpretation.

Workaround: Since the `-g` option is a very useful debugging option, move the variable declaration outside the body of the switch statement.

8) Compatibility Note: Local variable declarations of the form `char s[];` (as an array of unspecified length), may not behave as expected. Pre-ANSI compilers often expand such declarations into the equivalent of an extern declaration. MPW 2.0.2 C, and most modern compilers, treat this construct as the declaration of a new array, thus overriding any global declaration with the same name. In the following example

```
/* From file foo.c */
char s[255];

main()
{
        strcpy(s, "This is one heck of a mess");
        otherfunc();
}


/* From file bar.c */
otherfunc()
{
        char s[];

        printf("%s\n", s);
}
```

garbage is printed. As a local variable declaration, this declaration is incomplete since no length is specified or implied, so an ANSI C compiler will of course fail. This is obviously not a recommended programming practice, but if you are porting old C code you may encounter this usage.

Workaround: ALWAYS use the declaration `extern char s[];` instead.

9) Another instance where declarations of the form `type s[];` (as an array of unspecified length) may not behave as expected, is as a member of a `struct` or `union`. Pre ANSI compilers often expand such declarations into the equivalent of a pointer declaration. This construct is explicitly prohibited in ANSI C structures and unions. MPW 2.0.2 C on the other hand, issues no warning and treats this construct as the declaration of an array of length zero, which occupies no space in the `struct`. In the following example

```
typedef struct  ST1  {
        int array1[];        /* Zero length array or Ptr to array? */
        int array2[];
        int array3[];
}ST1;

main()
{
       ST1 s1;
       int i1, i2, i3;

       i1 = s1.array1[0];
       i2 = s1.array2[0];
       i3 = s1.array3[0];
}
```

the three fields of the `struct ST1` are located at the same memory location, and the assignments shown will actually copy garbage into the integers, since no space was allocated for even the first element of the arrays. Unfortunately, structures containing an array of unspecified or zero length as the final member are sometimes used to indicate a structure containing a variable length array. While this may be useful, it is not tolerated by ANSI C and thus is not a recommended programming practice. However, if you are porting old C code you may encounter this usage.

Workaround: ALWAYS use the declarations of the form `type *s;`, `type (*s)[];` , or `type s[1]` (depending on the intended meaning) in structures and unions instead.

10) The routines `_SetUpA5` and `_RestoreA5` described in the OS Utilities Chapter of *Inside Macintosh*, Volume II, are missing. Refer to Technical Note #208: Setting and Restoring A5 for two new routines which solve this problem.

11) Hint: Switch statements with large numbers of cases (over 100 or so) can trigger the appearance of the MPW Bulldozer cursor (signaling heap purging and compacting in progress), can cause "Out of memory" errors from the compiler, or at least take a very long time to compile.

Workaround: Large switch statements can be split up into smaller ones to avoid these symptoms.

Pascal Language

The following information applies to the Pascal compiler and its associated libraries shipped with the 2.0.2 version of MPW.

1) The Pascal compiler generates incorrect code when accessing the elements of `packed Boolean` arrays with more than 32767 elements. The generated code contains an `ASR.W` instruction to compute the byte offset into the array, this of course fails for numbers larger than 32767 ($7FFF). The following example, which zeroes bits far beyond the end of the array itself, illustrates the error:

```
PROGRAM PackArrayBug;

        VAR
          MyBits:      packed array[0..33000] of Boolean;

          PROCEDURE BadCode;
          VAR
               i: longint;
          BEGIN
               for i := 0 to 33000 do
                 MyBits[i] := false;
          END;

        BEGIN
               BadCode
        END.
```

Workaround: Don't use `packed Boolean` arrays with more than 32767 elements.

2) The Pascal compiler fails to detect the situation where a procedure name is passed as an argument to a routine expecting a function as an argument. The following example illustrates the error:

```
PROGRAM FuncArgBug;

          PROCEDURE ExpectsFunc(x: integer; function Visit(y1: longint;
                       y2: char;): Boolean);
          VAR
               result: Boolean;
               yc: char;
          BEGIN
               yc := 'A';
               result := Visit(x, yc);
          END;

          PROCEDURE FormalProc(y1: longint; y2: char;);
          BEGIN
               writeln(y1: 1, ' ', y2);
          END;

        BEGIN
               ExpectsFunc(5, FormalProc);
        END.
```

This type of problem typically leads to stack misalignment and spectacular crashes.

Workaround: Make certain that Pascal routines expecting functions as arguments are indeed passed functions.

3) The `-mc68881` option causes the Pascal compiler to generate incorrect code when calling external C language routines with floating point `extended` arguments. Apparently the compiler miscalculates the size of the `extended` argument so that it incorrectly removes the arguments from the stack. The following example, which can corrupt the local variables of its caller, illustrates the error:

```
FUNCTION E2S(f: Str255; x: extended): Str255;
   VAR
        dummy: integer;
        i:      integer;
        t:      Str255;

   FUNCTION sprintf(str: Ptr; fmt: Ptr; x: extended): integer; C; EXTERNAL;

   BEGIN
        t[0] := chr(0);
        f[ord(f[0]) +1] := chr(0);

        dummy := sprintf(@t[1], @f[1], x);

        i := 0;
        repeat
          i := i+1;
        until ((t[i] = chr(0)) or (i > 254));
        t[0] := chr(i);

        E2S := t;
   END.
```

The relevant portions of the generated code are:

```
LINK        A6,#$FDF0               ; LINK for local vars
MOVEM.L     D6/D7,-(A7)             ; Save Registers used here
...
...
LEA         $FF00(A6),A0            ; Get address of extended var
MOVE.L      -(A0),-(A7)             ; Push extended onto stack
MOVE.L      -(A0),-(A7)
MOVE.L      -(A0),-(A7)
PEA         $FF01(A6)               ; Push address of format str
PEA         $FDF1(A6)               ; Push address of target str
JSR         *+$0002                 ; JSR to sprintf
LEA         $0012(A7),A7            ; Pop args off stack
                                    ; (SHOULD pop off $14 bytes!)
MOVE.W      D0,D6                   ; Save function result
...
...
MOVEM.L     (A7)+,D6/D7             ; Restore registers used here
                                    ;  (Now they've been corrupted!)
UNLK        A6                      ; Correctly restores A7
MOVEA.L     (A7)+,A0
ADDQ.W      #$8,A7
JMP         (A0)                    ; JUMP back to caller
```

Notice that this code would have succeeded if the routine had not used the `D6` and `D7` registers for storage, and then restored them (incorrectly) before returning.

Workaround: When calling such a routine with the `-mc68881` option, isolate the call in a small subroutine or function that has no local variables, so that registers will not need to be saved and restored.

4) The `{$SC+}` compiler flag for short circuiting AND and OR operators can sometimes produce incorrect code. The following example does not work if `{$SC+}` has been used:

```
USES
  Memtypes,QuickDraw,OSIntf,ToolIntf,PackIntf;

VAR
  b1,b2,b3 : BOOLEAN;

Begin
  b1 := false;
  b2 := true;
  b3 := true;

  if not (b1 and b2) and b3 then
      SysBeep(40);
End.
```

Workaround: Don't use the `{$SC+}` compiler flag.

5) The Pascal compiler generates incorrect overflow checking code for `longint` valued arguments to the `Odd` function. The generated code contains a signed divide (`DIVS`) by 1 followed by a `TRAPV`, thus the overflow flag is set for values greater than $7FFF. The following example will fail with a `CHK` exception, unless the `{$OV+}` directive is removed:

```
{$OV+}
PROGRAM PascalOdd;

      VAR
            IsOdd: Boolean;
            longval: LONGINT;

      BEGIN
            longval := 123456;
            IsOdd := Odd(longval);
      END.
```

Workaround: Don't use the `{$OV+}` compiler flag if you pass `longint` values to the `Odd` function.

6) The Pascal compiler generates incorrect code when functions with `RECORD` type are used as the object of a `WITH` statement. This will only occur if the function is called at a level below the main program, and if the length of the `RECORD` type is 4 bytes or less. The generated code often contains an `LEA (A7)+`, an instruction, which is of course illegal. The following example demonstrates this unusual situation:

```
PROGRAM PasRecordValFuncBug;

  TYPE
    OurRecord   =
      RECORD
        a:    Integer;
      END;
```

```
  FUNCTION RecordValuedFunction: OurRecord;
  BEGIN
  END;

  PROCEDURE ContainsBadCode;

  BEGIN
    WITH RecordValuedFunction DO BEGIN  { This usage bad code. }
      a:=a;
    END;
  END;

BEGIN { PasRecordValFuncBug }
  ContainsBadCode;
  WITH RecordValuedFunction DO BEGIN  { This usage is okay. }
    a:=a;
  END;
END. { PasRecordValFuncBug }
```

Workaround: Don't use `RECORD` valued functions as the object of `WITH` statements.

Assembly Language

The following information applies to the Assembler and its associated libraries shipped with the 2.0.2 version of MPW.

1) There are no known outstanding bugs in the MPW Assembler.

Interface Libraries

The following information applies to the Toolbox and OS interface libraries shipped with the 2.0.2 version of MPW.

1) The glue for the Device Manager call `GetDCtlValue` [Not in ROM] described in *Inside Macintosh*, Volume III, is incorrect and will return an incorrect value for the handle to the driver's device control entry. The following is a corrected version of the erroneous glue found in the library file `Interface.o`:

```
;-----------------------------------------------------
;FUNCTION GetDCtlEntry(refNum: Integer) : DCtlHandle;
;-----------------------------------------------------
GetDCtlEntry PROC EXPORT
        MOVEA.L         (SP)+,A0        ;Get the return address
        MOVE.W          (SP)+,D0        ;Get the refNum
        ADDQ.W          #$1,D0          ;Change to a
        NEG.W           D0              ;  Unit Number
;===>   LSR.W           #$2,D0          ;==Shift in wrong direction!
        LSL.W           #$2,D0          ;Times 4 bytes/entry
        MOVEA.L         UTableBase,A1   ;Get address of unit table
        MOVE.L          (A1,D0.W),(SP)  ;Get the DCtlHandle
        JMP             (A0)            ;And go home
```

This error will affect C, Pascal, and assembly language users.

Workaround: Use the corrected glue for `GetDCtlValue`.

2) The glue for the register-based Vertical Retrace Manager calls `_SlotVInstall` and `_SlotVRemove` described in *Inside Macintosh*, Volume V, is incorrect. The following are corrected versions of the erroneous glue for these routines found in the library file `Interface.o`:

```
;---------------------------------------------------------------------
;FUNCTION SlotVInstall(vblTaskPtr: QElemPtr; theSlot: Integer): OSErr;
;---------------------------------------------------------------------
SlotVInstall PROC EXPORT
        MOVEA.L         (A7)+,A1        ; save return address
        MOVE.W          (A7)+,D0        ; the slot number
        MOVEA.L         (A7)+,A0        ; the VBL task ptr
        _SlotVInstall
        MOVE.W          D0,(A7)         ; save result code on stack
        JMP             (A1)            ; return to caller
        ENDPROC
```

```
;---------------------------------------------------------------------
;FUNCTION SlotVRemove(vblTaskPtr: QElemPtr; theSlot: Integer): OSErr;
;---------------------------------------------------------------------
SlotVRemove PROC EXPORT
        MOVEA.L         (A7)+,A1        ; save return address
        MOVE.W          (A7)+,D0        ; the slot number
        MOVEA.L         (A7)+,A0        ; the VBL task ptr
        _SlotVRemove
        MOVE.W          D0,(A7)         ; save result code on stack
        JMP             (A1)            ; return to caller
        ENDPROC
```

These errors will affect C, Pascal, and assembly language users.

Workaround: Use the corrected glue for `_SlotVInstall` and `_SlotVRemove`.

3) The glue for the register based Start Manager calls `_GetTimeout` and `_SetTimeout` described in *Inside Macintosh*, Volume V, is incorrect . The following are corrected versions of the erroneous glue for these routines found in the library file `Interface.o`:

```
;---------------------------------------------------------------------
;PROCEDURE GetTimeout(VAR count: INTEGER);
;---------------------------------------------------------------------
GetTimeout PROC EXPORT
;===>    CLR.W       -(A7)               ;===OOPS, selector in A0 not on stack
        SUBA.L      A0,A0               ;Put selector in A0, i.e. 0
        _InternalWait
        MOVEA.L     (A7)+,A1            ;Pop return address into A1
        MOVEA.L     (A7)+,A0            ;Pop location for VAR count
        MOVE.W      D0,(A0)             ;Stuff returned value into count
        JMP         (A1)                ;And go home

;---------------------------------------------------------------------
; PROCEDURE SetTimeout(count: INTEGER);
;---------------------------------------------------------------------
SetTimeout PROC EXPORT
        MOVEA.L     (A7)+,A1            ;Pop return address into A1
        MOVE.W      (A7),D0             ;Move count parameter into D0
;===>    MOVE.W      #$0001,(A7)         ;===OOPS, selector in A0 not on stack
        MOVEA.W     #$0001,A0           ;Put selector in A0
        _InternalWait
        JMP         (A1)                ;And go home
```

These errors will affect C, Pascal, and assembly language users.

Workaround: Use the corrected glue for `_GetTimeout` and `_SetTimeout`.

#201: ReadPacket Clarification

See also: The AppleTalk Manager

Written by: Mark Bennett August 1, 1988

This technical note clears up some confusion concerning the low-level function `ReadPacket`. This function is called by protocol handlers and socket listeners.

The documentation for `ReadPacket` on page 327 of *Inside Macintosh Volume II* states that MC680X0 register `D3` should be tested to determine if there was an error condition. This is incorrect. `D3` merely reflects the number of bytes left to be read and could be zero even though an error occurred. The correct test for an error condition after calling either `ReadPacket` and `ReadRest` is the `Z` (Zero) bit, which will be set if no error was detected and clear otherwise.

#202: Resetting the Event Mask

See also: *Inside Macintosh*, Volume II, The Event Manager

Written by: Chris Knepper August 1, 1988
Revised by: Chris Knepper December 1988

As of System 4.2 and Finder 6.0, applications which alter the event mask must now restore the original event mask when quitting.
Changes since August 1, 1988: Added a related MultiFinder anomaly.

In most cases, applications should **not** modify the system event mask, which means they should avoid calling `SetEventMask` and not alter the low-memory global `SysEvtMask`. Modifying the event mask is of no use to most applications, and the only situation in which an application might need to modify it is to detect key-up events. Only those developers creating applications which must detect key-up events need to know the information presented in this Technical Note. Other developers should avoid altering the system event mask at all costs.

Since the system event mask normally prevents key-up events from being posted, those applications which need to detect key-up events call `SetEventMask` during initialization to enable key-up events. This process might be as follows:

```
myMask := EveryEvent;
SetEventMask(myMask);
```

Applications which make this call during initialization, **must** save the event mask prior to calling `SetEventMask` and restore the event mask when quitting. Given the following definitions and declarations in MPW Pascal:

```
CONST SysEvtMask = $144;
TYPE  IntPtr = ^INTEGER;
VAR   saveMask: INTEGER;
      MaskPtr: IntPtr;
```

you save the event mask as follows:

```
{ set up our event mask pointer }
MaskPtr := IntPtr(SysEvtMask);
{ save the event mask }
saveMask := MaskPtr^;
```

and restore the event mask as follows:

```
{ restore the event mask }
MaskPtr^ := saveMask;
```

Finder Anomaly

When MultiFinder is disabled, users will notice a strange behavior in the Finder (versions 6.0 and later) after quitting applications which fail to restore the event mask. If an application failed to restore the event mask when quitting and had set the event mask to mask out mouse-up events, all mouse-up events would continue to be masked out, and the user would notice that the Finder no longer recognizes double clicks.

MultiFinder Anomaly

With the current Macintosh architecture, the interrupt handlers which service hardware interrupts call `_PostEvent` to post events in the event queue, so events are inserted in the queue unless they are masked out by `SysEvtMask`. If an event is masked out by `SysEvtMask`, then `_PostEvent` returns a `evtNotEnb` error and does not insert the event in the queue.

Applications normally retrieve events which have been successfully posted to the event queue by calling `_GetNextEvent` or `_WaitNextEvent`. If the event being retrieved is not masked out by the event mask (`mask`) which is supplied to these routines, then the routines will return `TRUE`.

Under MultiFinder, `SysEvtMask` is an application-specific global variable which is switched during context switches; therefore, under MultiFinder, if an application must alter `SysEvtMask`, it must also be prepared to handle all events, even those which normally would be masked out by `SysEvtMask`.

This anomaly occurs when MultiFinder switches out `SysEvtMask` during a "minor switch" (i.e., switching from foreground to background to service background applications). Interrupt service routines operating at interrupt time call `_PostEvent` to post events, and `_PostEvent` masks out events using the mask in `SysEvtMask` to determine whether or not to post the event. This scheme means that events are posted as they occur, regardless of whether the current value of `SysEvtMask` belongs to the foreground application or a background application; therefore, it is possible for the event queue to not contain key-up events, even though the foreground application enabled them, because the background application could mask them out.

A future version of the System Software will account for this problem by using the foreground application's event mask when events are posted (even if a background application is running when the interrupt service routine is called).

#203: Don't Abuse the Managers

See also: The Resource Manager
TextEdit
The List Manager
The Dialog Manager
Technical Note #141–Maximum Number of Resources in a File
Technical Note #34—User Items in Dialogs

Written by: Bo3b Johnson August 1, 1988

When using the various pieces of the Macintosh operating system there is a temptation to try to stretch the built-in Managers too far. Developers should be aware of the intended purpose of the various Managers and beware of using them for things that they were not designed to handle. If extended beyond their design goals, they will become slow and unwieldy.

Managers to avoid abusing, and the type of abuse:

1) The Resource Manager is **not** a database.
2) The TextEdit package is **not** a word processor.
3) The List Manager is **not** a spreadsheet.
4) The Dialog Manager is **not** a user interface.

No free database

After using the Resource Manager for a short time, its virtues become apparent: it is very flexible, it is easy to use, it gives disk based I/O with no extra calls, data can be extracted by either name or ID number, and the data is stored transparently so the caller can pretend the data is always available in a virtual memory fashion. With such wide ranging advantages, it would seem that the Resource Manager should be used for everything. It should be apparent that the TANSTAAFL (There Ain't No Such Thing As A Free Lunch) philosophy applies to the Resource Manager as well. If overextended, the Resource Manager will become slow and unusable.

The Resource Manager is not a database, nor is it a good way to store user data. Although it can be used to store very small amounts of data, such as configuration data, and features some of the same characteristics of databases in general, the Resource Manager is a specialized tool designed specifically for the types of things that the Macintosh System needs. Its main virtue for system use is that a large variety of data can be stored on disk, and accessed when needed. This is a primitive form of virtual memory

which extends the power of the system beyond what the RAM supplies. Remembering that the Resource Manager was written in an era of 128K RAM, it should be apparent that it is optimized to use as little RAM as possible.

The Resource Manager uses a simple data structure for accessing the data in the file. Examining the Resource Manager file format can show some of the tradeoffs expected. For instance, there is a linearly accessed table which describes all of the possible resource types that are in the current file. Without too much thought it should be apparent that if a file is created with thousands of different resource types then access to those resources will be slow. The reason? Each access requires scanning a linear array. There is no hashing technique used on the resource types.

There is a similar linear table for the resource IDs themselves. Based on the previous discussion it should also be apparent that if there are thousands of resources of a specific type that the access time will become much larger. It will be imperceptible on a single access of a resource, but for thousands of accesses to the resource file the time spent traversing the linear list will impact the overall speed of the program. The user will not be pleased.

Increasing the slowness by having too many resources as well as too many types will encourage the user to file the program in a ground based circular storage facility.

As stated in Technical Note #141, there is a limit of about 2700 resources in a given file due to the way the resources are stored. The performance penalty will arrive sooner, and the dividing line for where it is "too slow" is a personal preference. As a rule of thumb, if the program has the ability to store more than about 500 resources total (both IDs and types), then consideration should be given to using the Data Fork instead. In particular, if the program allows the user to create data files, do not use the Resource Manager to store the user data. The users will always overextend the use of a program. Plan for it, and avoid making obviously bad decisions. For large amounts of data, the File Manager is the place to look. If the program wants to allow simultaneous (multi-user) access with read and write privileges to data files, then do not use the Resource Manager. Because it caches data, the Resource Manager cannot be relied upon as a multi-user database – even for small amounts of data. This is because there is no way to tell the Resource Manager its cache is invalid.

Don't be fooled by a convenient interface. The Resource Manager is not a database, nor is it a file system.

Words to live by

Looking at the TextEdit package can give the impression that there is a full featured word processing system built in. This is even more true now that TextEdit has been extended to support various styles and fonts. Unfortunately, appearances are deceiving, and TextEdit is not up to the job of being a word processor. Looking through the documentation shows that there is a 32,767 character limit on the text in a TextEdit record. The `teLength` is defined as an `Integer`. Another more subtle limit is the drawing limit of the rectangles surrounding the text. The `destRect` and `viewRect` both surround the complete TextEdit record. Using some rather rough approximations, there

is an upper limit of about 40 pages of text that can be supported in the QuickDraw rectangle. This is quite a lot for some applications, but is not very many when looking at the job typically required of a word processor. Users do not enjoy breaking their documents into multiple pieces.

There are some other programmatic limitations, not the least of which is performance. TextEdit will become quite sluggish with large blocks of data. After 2,000-4,000 characters have been stored in a TextEdit record, the performance will have slowed to an unacceptable level. It is notable that the `lineStarts` array is a linear array of offsets into the edit record. If the data towards the end of the data record (high in the record) changes, the offsets have to be changed. This can involve updating thousands of `Integer` offsets for every character typed. If the different font, size and style information is tacked on top of all that, the performance can be expected to suffer with large blocks of text. Make no mistake about it, a full Macintosh style word processor is not an easy thing to write. TextEdit was not designed to handle large documents. It was designed as a simple field editor for the Dialog Manager, and extended from there. It was never intended to handle the large jobs expected of a word processor.

In order to perform the operations required of a word processor it is necessary to use QuickDraw extensively. The expected Macintosh selection approach with autoscrolling, typing over selected text, cut/copy/paste, and so on are best implemented using QuickDraw directly. How the text is stored internally is the primary determining factor on how the word processor will perform.

Don't be fooled by how easy it is to implement simple editing in an application. TextEdit is not a word processor.

Checking lists twice

The List Manager appears to be a cell oriented display tool, allowing the easy creation of a spreadsheet interface using system calls. The rich interface to the manager makes it easy to handle arbitrary lists of data. Or does it? Although the List Manager is very flexible, easy to use, and general enough to handle graphic elements, its performance becomes unacceptable with relatively modest amounts of data. A one-dimensional list (like the files list in StdFile) can be done very well using the List Manager, but with several thousand items in the list, the performance may not be sufficient. This rarely happens in StdFile of course, and StdFile was the father of the List Manager. Here again, the tool was designed with a specific concept in mind, not to be the ultimate tool for handling any possible arbitrary data. A two-dimensional list of data will become too slow to use with an array as small as 10x100. This can hardly be expected to satisfy the user of a spreadsheet, since one "power" criteria is always the number of cells available.

Why so slow? As above, examining the data structures used by the List Manager can tell a lot about the expected performance and limitations. Notably the `cellArray` used to offset to each cell's data is an old friend, a linear array of `Integer` offsets. It should come as no surprise that inserting or deleting data from the middle of this array is slow. In order to do those functions the List Manager has to update the `Integer` offsets in the array each time. It has to step through each element on the linear array of offsets which will take some time on several thousand elements.

The `maxIndex` field of the ListRec is also notable since it is an `Integer` as well. The lists of data can be no more than 32K bytes in size, which could be somewhat limiting to a user.

In addition, the List Manager is very general purpose, making it necessary for it to protect itself from bad data whenever possible. It needs to check the bounds of any rectangles it uses for example. It tries to minimize drawing out of bounds, so it checks each cell as it is drawn to be sure that it is on screen. Extra validity checks take some small, but finite, time. As the number of elements grows, the time adds up until it becomes a performance problem. Another limitation brought out by the data structure is the `listDefProc`, the list definition procedure. Since the List Manager is designed to be as general purpose as possible, it was necessary to add the ability to plug in a new defproc. This has ramifications for speed, however, since all drawing has to go through the bottleneck of the defproc. It won't cost much each time, but it will add up over a large number of cells.

In order to get high performance out of this type of display, it is generally necessary to have as much precalculated as possible. This usually means having data structures which maintain themselves as much as possible, and which do not require changing anything outside of their single cell, thus avoiding impacting the entire display. Linear arrays don't come under this category, since any change impacts all the other cell data in the list. To create a high performance spreadsheet it is usually necessary to go to the QuickDraw level inside of a standard window. It is not typically necessary to be fully general for a specific type of data, so the performance can be improved merely by knowing the type of data expected. To handle large lists of data, the data should be stored in powerful data structures, and displayed with custom routines that know the best way to draw the data.

Don't be fooled by the richness and general purpose interface to the List Manager. The List Manager is not a spreadsheet.

Dialog with the devil

The Dialog Manager is very attractive. It looks like it will handle windows automatically with no programmer intervention, and can handle a wide variety of elements. It seems to handle controls, static text, editable text, and provides a way to display graphic elements as well. It must be the best possible world since the interface is very straightforward, and so much is done for the caller. At last, a superbly general purpose manager that can be used for any interface. Suddenly, reality rears its ugly head again, and it is interesting to note that this free lunch actually requires **more** work than doing the same job using the Window Manager, QuickDraw, TextEdit, and the Control Manager. Why? There is a hidden cost in terms of getting the Dialog Manager to do exactly the desired task. Here again, if the end result is supposed to be a simple dialog with a few controls, the Dialog Manager is suited to the job. That is what it was written to do. It was not designed as a way to handle the full interface for applications.

As an example of a hidden cost, what if the interface requires that the program be able to handle a disk inserted event? If this is part of a `ModalDialog`, that requires passing a

special `filterProc` to the dialog when it is called. It is now necessary to fully understand how the proc gets called, what is legal, and what the proc is required to do. That may not be too hard, but it is time spent on something that has nothing to do with getting the job done; it is only time spent understanding how the Dialog Manager works.

Another example is adding something to a dialog which requires special setup and update routines. Here again, it is not too hard to figure out, but it is time spent trying to tell the Dialog Manager what should be done. There are literally hundreds of these special cases and tough, small problems when trying to extend a dialog past a simple interface. Hundreds of Mac programmers have wasted hundreds (thousands?) of hours finding ways to coerce the Dialog Manager into running a window in a special way.

How about adding a special control to a dialog? Seems straightforward... How about making it modeless instead? How about moving some items in the dialog off screen? How about moving an EditText item off screen? How about wanting to change the dialog template before the dialog is used? How about all of the above all at the same time?

How about skipping it and using the Window Manager instead?

There are a number of performance penalties for large dialogs as well. A dialog with 50 radio buttons will be unacceptably slow. It should be noted that the Dialog Manager cannot know the desired purpose of the buttons, so it cannot set the button, nor clear another in the same set. In order to implement the actual radio button aspect of a set of controls, it has to be done by the calling program. At this point, the only thing the Dialog Manager is handling is the creation and drawing of the controls, which can easily be done with `GetNewControl` and `DrawControls`. The Dialog Manager actually gets in the way of a more complex interface. Looking into the data structures shows that the list of items in a dialog is a linear list. Also of note is that there are no offsets to the various items! This is significant because it means that the Dialog Manager has to drive through the entire list of items for every single operation it performs. If it gets an update event it has to traverse the list. If it gets a mouse event it has to traverse the list. This cannot be expected to be fast with 100 items.

Another performance problem for some programmers is the simple drawing scheme used by the Dialog Manager. If a dialog has some items that are offscreen, they get drawn during update events anyway. The Dialog Manager will traverse the list and draw each item, whether it is on screen or not. This comes from the original design of the Dialog Manager, in that it was never intended to handle hundreds of items, or items off screen.

Some rules of thumb: If there are more than 20 items in the dialog it should be a standard window. If a complicated control like a scroll bar is needed, it should be a standard window. If there are items offscreen, it should be a standard window. If there is a pictorial indicator like a progress indicator, it should be a standard window. If it is a modeless dialog it should be a standard window. If any of the items are movable in the dialog, it should be a standard window. If it is necessary to use a `filterProc` to add functionality, it should be a standard window. If in doubt, it should probably be a standard window.

Handling a dialog with the Window Manager is very straightforward, much more so than trying to get around the Dialog Manager. There is the standard main event loop, and a conventional case statement to handle the events of interest. If there are controls in the window, they are easily handled with Control Manager calls. Any special items can be added to the case statement with no tricks. Overall there is more code to write, but the code is much less complex (read as: easier to figure out, easier to debug, easier to maintain). In addition, when extra items have to be added to the window, there is an easy-to-find, logical place to add the code. With the Dialog Manager there may be hidden difficulties.

The Dialog Manager is very powerful, but to use the power it is necessary to use all sorts of hooks, procs, special items, and special calling sequences. As expected, only the interfaces to these things are described in *Inside Macintosh*. The sequence of events is the costly part. For an example of how to add a `userItem` to a dialog, examine Technical Note #34. Note that it is not particularly simple to understand. Contrast that with the `FillRect/FrameRect` calls in the code that handles update events in a normal window.

The Window Manager is more powerful than the Dialog Manager. The Dialog Manager uses the Window Manager. The Window Manager is much more straightforward to use since it follows the conventional Macintosh event model. That model is easier to understand and easier to extend. There are more calls to make, but the overall use is much simpler. There are very few special tricks needed to make any conceivable interface in a window.

Don't be lured in by the "powerful" Dialog Manager calls, tricky hooks, and filter procedures. The Dialog Manager is not a user interface.

#204: HFS Tidbits

See also: The File Manager
Technical Note #77–HFS Ruminations
Technical Note #102–HFS Elucidations

Written by: Dave Burnard August 1, 1988

This Technical Note describes two poorly documented features of the File Manager.

Always Set ioFVersNum to Zero

When making a File Manager call which uses a `CInfoPBRec`, or the `fileParam` or `ioParam` portion of either a `ParamBlockRec` or an `HParamBlockRec`, you should set the `ioFVersNum` field to zero. File version numbers are an artifact of MFS and are not supported on HFS volumes or by the Resource Manager or Standard File Package. In fact, the `ioFVersNum` field is ignored when accessing an HFS volume. Unfortunately, when accessing an MFS volume, the version number **is** still used, and should be set to zero.

A little known fact that can lead to difficulties, is that many of `PBHxxxx` File Manager calls "fall through" to their `PBxxxx` counterparts when accessing MFS volumes. For example, although the interface to `PBHOpen` in *Inside Macintosh Volume IV* does not indicate that the `ioFVersNum` field is used, when opening a file on an MFS volume, `PBHOpen` falls through to `PBOpen` which **will** use the version number. Unless `ioFVersNum` is explicitly zeroed this can lead to unexpected "file not found" errors.

Incorrect PBSetVInfo description

The interface to `PBSetVInfo` as described in the File Manager chapter of *Inside Macintosh Volume IV* **incorrectly** indicates that the volume allocation clump size, the minimum number of volume allocation blocks added to a file when its length increases, can be set with the `ioVClpSize` field. The `ioVClpSize` field is actually ignored by `PBSetVInfo`.

Macintosh
Technical Notes



Developer Technical Support

#205: MultiFinder Revisited: The 6.0 System Release

Revised by: Andrew Shebanow, Jim Reekes, & Dave Radcliffe
Written by: Dave Burnard

April 1990
August 1988

This Technical Note describes several new features found in MultiFinder 6.0 and answers a few more commonly-asked questions.
Changes since December 1989: Added a warning to the section on `childDiedEvents` about distribution of MultiFinder 6.1bx.

How Can I Tell If MultiFinder is Present

Once again, you cannot. Previous Technical Notes discuss how to check for the new services available with MultiFinder (i.e., `_WaitNextEvent` and the temporary memory allocation calls).

Currently, since an application cannot tell if MultiFinder is present, the application also cannot know how a sublaunch will behave (see Technical Note #126, Sub(Launching) from a High-Level Language). Unfortunately, the two possible sublaunch behaviors are radically different; with MultiFinder the `_Launch` trap returns to the application and without MultiFinder it does not. For most applications, however, these differences in sublaunch behavior should not matter. Hopefully, the `_Launch` trap will be improved in a future System Software release.

_WaitNextEvent is Always Available

In System 6.0 and later, `_WaitNextEvent` is present whether or not MultiFinder is present. Calling `_WaitNextEvent` without MultiFinder installed is virtually identical to calling it with MultiFinder installed. Your application can still "sleep" for a specified time and be notified if the cursor location is outside a specified region. The only difference when MultiFinder is not installed, is that your application is not suspended or resumed. If your application requires System 6.0 or later, DTS recommends calling `_WaitNextEvent` instead of `_GetNextEvent` in your main event loop.

_MFTopMem

The *Programmer's Guide to MultiFinder*, which is distributed through APDA, incorrectly documents `_MFTopMem` on page E-1. It does not return a pointer to the top of your application's memory partition as it is documented. It does, however, return a pointer to the top of the addressable RAM space in the machine, and is documented correctly on page 3-15 of the manual. Note that earlier releases of this manual referred to this call as `_MFMemTop`.

MFTempHandles Are Not Handles

The MultiFinder temporary memory allocation call, `_MFTempNewHandle`, currently does not return a "true" `Handle` in the sense that it can be used interchangeably with a `Handle` obtained from a call to `_NewHandle`. Specifically, you cannot pass a `Handle` obtained from a call to `_MFTempNewHandle` to any Memory Manager routine or Toolbox routine which, in turn, passes it to the Memory Manager (either directly or indirectly). Like a true `Handle`, however, you can still dereference a `Handle` obtained from `_MFTempNewHandle`. You should treat a `Handle` from `_MFTempNewHandle` in the same way you would a fake `Handle` (i.e., a `Handle` not obtained from the Memory Manager—see Technical Note #117, Compatibility: How & Why). This restriction on the use of MultiFinder temporary memory may not apply in future System Software releases.

Mouse-Moved Event Confusion

There has been some confusion over the `mouseRgn` parameter to `_WaitNextEvent`, and under what circumstances it returns a mouse-moved event. Most of the confusion is caused by the word "moved." Many applications have assumed that mouse-moved events are generated only when the mouse actually leaves the mouse region. In System 6.0 and later, `_WaitNextEvent` returns a mouse-moved event whenever the cursor is outside the mouse region. Thus, when an application receives a mouse-moved event, it should compute a new mouse region based upon the new cursor location before calling `_WaitNextEvent` again, otherwise `_WaitNextEvent` continues to return mouse-moved events until the user moves the cursor back inside the mouse region or until a new mouse region is specified.

New MultiFinder Features

Open Document and Quit

In System 6.0 and later, MultiFinder adds the ability to open application documents from the Finder when the owner application is already open. For the moment, MultiFinder accomplishes this by simulating a mouse-down event in the application's menu item for opening files. The application usually responds by calling `_SFGetFile`, which MultiFinder short circuits into returning the document opened in the Finder layer. This is similar to the way that MultiFinder triggers applications to quit when the user selects Shut Down or Restart from the Finder's Special menu.

In future System Software releases, this mechanism will probably change to a more straightforward method of notifying the application that it needs to open a document or to quit.

How does MultiFinder find the Open item? By default, MultiFinder looks for a File menu with an item named Open..., Open ..., Open..., etc. Of course, some applications do not have a File menu or they name their Open item something different (i.e., Open Document). To compensate for this difference, MultiFinder **first** looks in the application's resource fork for `'mstr'` or `'mst#'` resources in the range of 100-103. An `'mstr'` resource has the same format as an `'STR '` resource (a Pascal string) and contains the name of the menu or menu item for which MultiFinder should look. An `'mst#'` resource has the same format as an `'STR#'` resource (a list of Pascal strings) and contains a set of names for the menu or menu item for which MultiFinder should look. MultiFinder uses this same mechanism to locate the application's Quit command. Table 1 documents these resource IDs and their meanings.

Resource ID	Meaning
100	Name or names of the menu containing the Quit command.
101	Name or names of the menu item or items corresponding to the Quit command.
102	Name or names of the menu containing the Open command.
103	Name or names of the menu item or items corresponding to the Open command.

Table 1–Resource IDs and Meanings

As always, be careful to avoid any "clever" tricks that rely upon this information; MultiFinder will **not** always work this way.

Additions to the 'SIZE' Resource

The `'SIZE'` resource has four new flags (`onlyBackground`, `getFrontClicks`, `acceptChildDiedEvents`, and `is32BitCompatible`) which communicate information about an application to MultiFinder. Figure 1 illustrates the locations of these new flags. Setting **both** the `onlyBackground` flag **and** the `canBackground` flag informs MultiFinder that an application is a "faceless background task," that is, it has no user interface (i.e., no windows and no ports) and should only be run in the background. An example of a faceless background task is the System Software application Backgrounder.

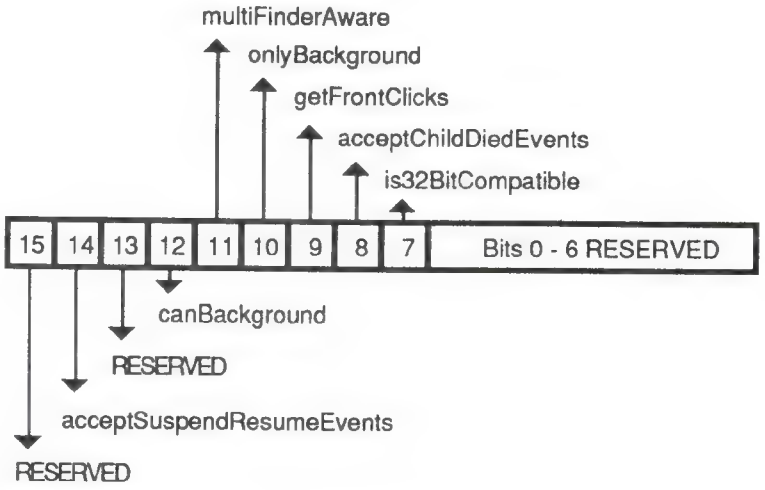


Figure 1–'SIZE' Resource Flag Bits

An application can set the `getFrontClicks` flag if it wants to receive the mouse-up and mouse-down events when the user brings the application's layer to the front. Typically, the user merely wants to bring an application to the front, so it may not be desirable to move the insertion point or start drawing immediately after coming to the foreground. If `getFrontClicks` is set, the mouse click is passed to the application. If `getFrontClicks` is set and a click is made in the content region of the background application's frontmost window, then the application receives a click in the content region of that window.

Clicking on a window that is behind another window within the same layer causes the usual event processing (i.e., the mouse-down event is visible to the application), for which the application calls `_SelectWindow`, to bring the window forward. This is true whether or not the bit is set. Ordinarily, these events are not passed to the application, so setting the `getFrontClicks` flag is

usually not appropriate. The Finder, however, is one example of an application which has the `getFrontClicks` flag set.

The `acceptChildDiedEvents` flag is used by SADE to get notification when an application it launched quits or crashes. A `childDiedEvent` is another MultiFinder `app4Evt` with a message field of the event record which Figure 2 illustrates.

$FD	Status	Reserved	Reserved

Figure 2–Message Field of childDiedEvent

Note: Your application does not receive `childDiedEvent` events unless the user is running MultiFinder 6.1b7 (shipped with SADE 1.0) or 6.1b9 (available on AppleLink in Developer Services: Macintosh Developer Technical Support: Tools: SADE MultiFinder). The MultiFinder which comes with System 6.0.x (and earlier) does not send these events. Your application should not depend on these events for its operation—they are documented for debugger use only. In addition, developers may **not distribute** MultiFinder 6.1bx to customers, even if licensed to distribute Apple's Macintosh System Software.

The `Status` parameter in the message field is a system error code if the application crashed or zero if it quit normally. The `where` field of the event record contains the process identifier (pid) of the quitting process. The `_Launch` trap returns the pid of the newly created application in `D0` if the call to `_Launch` succeeds (if `D0` is negative, it contains an OS error code).

Note: Future versions of System Software may operate only in 32-bit mode on machines with 68020 or newer CPUs, and applications which are not 32-bit clean will not function correctly on these machines.

The `is32BitCompatible` bit will be used in future systems to warn users that running an application which does not have the bit set may **crash** their system, if it is running in 32-bit mode. Developers should **not** set this bit unless they have thoroughly tested their applications on a 32-bit system. Currently, the only 32-bit system available for testing is A/UX, so running under A/UX should be considered the "litmus test" for 32-bit compatibility until newer System Software is available. Note, however, that the `is32BitCompatible` bit does **not** have to be set to run an application under the current version of A/UX.

Further Reference:

- *Programmer's Guide to MultiFinder* (APDA)
- MultiFinder Development Package (APDA)
- Technical Note #117, Compatibility: How & Why
- Technical Note #158, Frequently Asked MultiFinder Questions
- Technical Note #177, Problems with _WaitNextEvent in MultiFinder 1.0
- Technical Note #180, MultiFinder Miscellany
- Technical Note #212, The Joy Of Being 32-Bit Clean

Macintosh
Technical Notes

Developer Technical Support

#206: Space Aliens Ate My Mouse (ADB–The Untold Story)

Revised by: Rich Kubota October 1991
Written by: Cameron Birse August 1988

This Technical Note explains how the Apple Desktop Bus (ADB) works on the Macintosh. This Note covers the boot process, driver installation, ADB Manager run-time behavior, use of ADB Manager calls, and answers commonly asked questions.
Changes since February 1990: Added description of the boot process to include detail on how the ADBS resource gets called by the System, added detail to 2 of the answers in the Q&A section, and added sample completion routines for the ADBOp function.

Boot Process

During the boot process, the ADB Manager finds all the devices on the bus and resolves any address conflicts. An address conflict is defined as two or more devices with the same original (default) address. A good example of this conflict is a mouse and a graphics tablet that are both at address 3 (relative device). The ADB Manager resolves these address conflicts as described in Appendix B of the ADB Specification (Apple Drawing #062-0267-E) and the Q & A section of this document.

After the address resolution, the devices which have been "moved" due to address conflicts are addressed, starting from the highest unused soft address and working down. The system now loads and executes all the resources of type `'ADBS'` that match the devices on the bus (by original address).

Once all the ADB service routines are installed, the ADB transceiver (microcontroller) chip starts polling the active device. The active device is defined as the last device to send data. Since the mouse (pointing device) is the most likely device to have data ready at any given time, it defaults as the active device after startup.

The transceiver polls the active device (approximately every 10-16 milliseconds, do **not** depend on this interval), with a Talk R0 command. If the active device has new data, it can respond with it, and if it does not, it just times out. If any other devices have data to send, they can assert SRQ (refer to Figure 5 of the ADB Specification) at the end of the Talk R0 command. When the host detects an SRQ, it begins polling all addresses with a Talk R0 command until one returns data. That device then becomes the active device.

Devices have no way of knowing if they are the "active device". The algorithm for a device with data ready to send is as follows:

- Wait for a Talk R0 command.
- If the Talk R0 is for you, then return the data.
- If the Talk R0 is not for you, wait for the end of the command, and assert SRQ.
- If the Talk R0 is addressed to you, then respond with your data.

Now that a device has been polled, the host retrieves the data from the bus and calls the service routine installed for that device (service routines are installed by calling `_SetADBInfo` and are maintained by the ADB Manager). The system passes pointers to the service routine itself, its data area, and the data received from the device, as well as the ADB command byte that caused the routine to be called.

Normally, the service routine does not need to use the `_ADBOp` call to retrieve data. The ADB "philosophy" assumes that register zero of a device is the main data transmission register. Since register zero is automatically polled by the system, there should be no need to call `_ADBOp` from the service routine. Typically, `_ADBOp` is used to set modes of a device, or to interrogate the device for status—the sort of things that should not need to be done more than once or twice during normal operation.

It is important to note that ADB service routines are called at interrupt time, which means that they must follow all the rules regarding code that executes at interrupt time. (See *Inside Macintosh* references to VBL tasks and Device Manager I/O completion routines.)

Installing an ADB Service Routine and Optional Data Area

At boot time the system searches for `'ADBS'` resources in the System file. The system matches desktop bus devices by their original address to an `'ADBS'` resource (i.e., if the machine has a device that responds at address 4, the system looks for an `'ADBS'` resource with ID=4). The limitation of this method is that there can only be one `'ADBS'` resource for each address on the bus.

When the system finds these resources, it loads, detaches, and executes them. The System loads in each ADBS driver with a `_GetResource` call. If successful, the System calls _DetachResource on the handle to the ADBS driver. The registers are set up as described below, and a JSR (A0) call is made to execute the resource. It is the responsibility of the driver to dispose of itself if a failure occurs.

If there is insufficient memory in the System heap to load the resource, the ADBS will not be executed, and the System continues on to the next ADBS resource. For this reason, the size of the ADBS resource should be kept as small as possible. This condition should only occur under System 6.0.x and earlier.

A typical `'ADBS'` resource allocates space in the system heap for its service routine and, optional, data area. Next, it moves the service routine into the allocated space and initializes the data area, if necessary. This code should also install an `_ADBReInit` preprocessing routine to deallocate the memory used by the service routine (*Inside Macintosh* V-367).

When the system loads and executes an `'ADBS'` resource, it passes the following parameters:

`A0` = Address of `'ADBS'` resource in memory.

D0 = ADB device address (0-15). This address may be different than the "original address," since it occurs after address resolution.
D1 = ADB Device Type (same as the handler ID)

With this information, the 'ADBS' code can call _SetADBInfo to install the service routine and data area. The installer should make sure the handler ID (Device Type) is the one it expects.

Note: Previous versions of this note advised using an 'INIT' resource as an alternative method for installing ADB service routine. Apple no longer advises this method. ADB service routines should only be installed by an 'ADBS' resource located in the System file (see MacDTS Sample Code #17, TbltDrvr for an example). 'INIT' resources and application-based installation methods do not work on the Macintosh Portable, because the bus and ADB Manager may be re-initialized after waking up. Part of the re-initialization process loads and executes the 'ADBS' resources associated with the devices present on the bus. If a service routine is installed using the 'INIT' or application method, it does not get re-installed when the Macintosh Portable wakes up.

General ADB Manager Run-Time Behavior

Since the implementation of the ADB Manager on Macintosh CPUs has varied slightly, it's useful to know what behavior to expect, and what not to depend upon. System Tools disks after 6.0.4 make the ADB Manager consistent on all Macintosh models.

Address Resolution

It is important that devices implement the collision detection and address moveability to prevent possible conflicts between devices that have the same default address.

Auto-Polling

All devices on the bus should expect, and be able to handle, being auto-polled. If they do not have a reason to respond, they should simply ignore the poll (time out). If the ADB Manager auto-polls a device which has no service routine installed, it simply throws away any data it may have gotten from the device. The Macintosh SE, SE/30, II, IIx, and IIcx implementations of the ADB Manager only auto-polls devices that have previously responded to an auto-poll, or that have requested service (by asserting SRQ). In addition, the Macintosh IIci and Portable implementations also auto-poll the last device addressed by an _ADBOp command—regardless of whether they have a service routine installed (via the _SetADBInfo call).

System Tools disks later than 6.0.4 patch the ADB Manager so that if the device does not have a service routine installed (with _SetADBInfo), it should not get auto-polled. In the unlikely case that SRQ is active and none of the devices with routines installed respond, the ADB Manager polls all devices (every address) trying to clear the SRQ. This case is why all devices on the bus should expect, and be able to handle, being auto-polled.

Note: _ADBOp commands always have priority over auto-polling and SRQ polling. Whenever there are pending _ADBOp commands in the command queue, they are executed before the host resumes auto-polling. Therefore, applications should not issue _ADBOp commands repeatedly, keeping the command queue full. Doing so results in effectively "locking up" the mouse and keyboard, which rely upon the auto-polling and SRQ polling to provide user input.

SRQ Polling

Since the ADB Manager may be polling any device on the bus when an SRQ happens, an application should not rely upon the sequence in which it polls devices. Instead, simply remember that the ADB Manager polls all devices, in turn, on the bus until SRQ is no longer asserted. After SRQ has been satisfied, the ADB Manager begins auto-polling the last device from which it got data.

ADB Manager Bugs 'n Fixes

_ADBOp Talk Command

Through System Software 6.0.4, there is a bug in the Macintosh SE, SE/30, II, IIx, and IIcx implementations of the ADB Manager where the count byte returned by an `_ADBOp` Talk command that timed out is not set to zero to reflect that no bytes were transferred. In addition, the two bytes following it are both $FF (these should be ignored). On the Macintosh IIci and Portable implementations, the count byte for a time out is zero, and any bytes which follow it should, of course, be ignored. This bug is fixed in System Tools disks after 6.0.4.

_ADBOp Listen Command

There is also a bug in the Macintosh SE, SE/30, II, IIx, and IIcx implementations where the number of bytes transferred was one off from the supplied count byte. On the Macintosh IIci and Macintosh Portable implementations the number of bytes transferred is what the count byte specifies. This bug is fixed in System Tools disks after 6.0.4.

There is a bug in the Macintosh Portable where all Listen commands with a data count greater than six send garbage in the seventh and eighth bytes. This bug is fixed in System Tools disks after 6.0.4.

_ADBOp Completion Routines

There is yet another bug in the Macintosh SE, SE/30, II, IIx, and IIcx implementations of the ADB Manager where the completion routines passed to the `_ADBOp` routine are not always called. This bug is not present in the Macintosh IIci and Macintosh Portable implementations and is fixed in System Tools disks after 6.0.4.

_ADBReInit

Inside Macintosh, Volume V states that `_ADBReInit` should be called with a device is added to the bus while the system is running. This statement is misleading. Do **not** attach devices of **any** kind to a Macintosh while the power is on. If there is a device that can be added to the bus via software (i.e., a device is already attached, and an additional "virtual" device can be added under software control), then it may be useful to call `_ADBReInit`, but it is not absolutely necessary. Devices can be added by simply installing a service routine for the appropriate address using the `_SetADBInfo` call.

However, if you do plan on using `_ADBReInit`, then you should know about the following bug with keyboard layouts (`'KCHR'` resources) other than the standard U.S. layout (`ID = 0`). Most international systems use alternate `'KCHR'` resources and may permit switching between them. On these systems, when `_ADBReInit` is called, it does not reinstall the current `'KCHR'` resource, but instead reinstalls the default U.S. `'KCHR'` resource (`ID = 0`). This problem is

evident on the Macintosh Portable, since it may call when it wakes up. This bug is fixed in Systems Tools disks after 6.0.4.

Users can fix this problem by toggling the keyboard mapping selection in the Control Panel. From an application, one could install an `_ADBReInit` post-processing routine (in the low-memory variable `jADBProc`, see *Inside Macintosh*, Volume V, The Apple Desktop Bus, pp. 367-368), which reinstalls the correct `'KCHR'` resource using the Script Manager `_GetEnvirons` and `_KeyScript` calls (see Technical Note #160, Key Mapping) after a call to `_ADBReInit`.

```
KeyScript(INTEGER(GetEnvirons(smKeyScript)));
```

This code makes a `_KeyScript` call with the current keyboard script (as described in the `'itlb'` system resource). The 'KCHR' and 'SICN' IDs for that script are already setup in the 'itlb' resource and in the appropriate script's local variables. For an example of a `jADBProc`, see MacDTS Sample Code #17, TbltDrvr.

Answers to Commonly-Asked ADB Questions

Question: I need information on developing an Apple Desktop Bus product. (Hey, that's not a question!)

Answer: Apple's Desktop Bus and ADB Device Specifications are a licensable product available through Software Licensing. For more information, contact:

Apple Software Licensing
Apple Computer, Inc.,
20525 Mariani Avenue, M/S 38-I
Cupertino, CA, 95014
(408) 974-4667
AppleLink: Sw.License
Internet: Sw.License@AppleLink.Apple.com

Additional ADB references are as follows:

Macintosh
Inside Macintosh, Volume V, The Apple Desktop Bus
Macintosh Family Hardware Reference

Apple II
Apple IIGS Hardware Reference Manual

Desktop Bus
Apple IIGS Firmware Reference Manual

General
Baum, Peter. "Boarding the Bus," *MacUser*, July 1987, p. 142.
"An Overview of Apple Desktop Bus,"
Call A.P.P.L.E., June 1987, p. 24.

Question: I would like to extend the keyboard cable for my Macintosh. How can I do this, and how can I make the extension?

Answer: The ADB specification states the maximum length of all cables on the Desktop Bus is five meters. If you wish to use longer cables than those supplied with the ADB

device, Kensington MicroWare (800) 535-4242, Monster Cable (800) 331-3755, and Data Spec (800) 431-8124 all supply them.

Disclaimer: This listing for Kensington MicroWare, Monster Cable, and Data Spec neither implies nor constitutes an endorsement by Apple Computer, Inc. If your company supplies these cables and you would like to be listed, contact us at the address in Technical Note #0.

Question: How can I use the LEDs on the Apple Extended Keyboard?

Answer: Using the LEDs on the extended keyboard involves the `_ADBOp` call. Once you determine that you have an extended keyboard (with `_CountADBs` and `_GetIndADB`), then register 2 of the extended keyboard has the LED toggles in the low 3 bits of the second data byte.

Therefore, you would do a Talk to register 2 to have the device send you the contents of register 2, manipulate the low three bits to set the LEDs, and then pass the modified register 2 back to the device with a Listen to register 2 command.

The Apple Extended Keyboard has an ID of 02 and a device handler ID of 02, while the Apple Standard Keyboard has an ID of 02 and a device handler ID of 01.

Note: At this point it is not clear what Apple has in mind for these LEDs, so you are using them at your own risk.

Question: I am confused about the service routines and data areas passed in the `_ADBOp` call. What does it all mean?

Answer: That 's a good question.

```
FUNCTION ADBOp (data:Ptr; compRout:ProcPtr; buffer:Ptr;
                commandNum:INTEGER) : OSErr;
```

`data`	is a pointer to the "optional data area". This area is provided for the use of the service routine (if needed).
`compRout`	is a pointer to the completion or service routine to be called when the `_ADBOp` command has been completed -i.e sent to the device. Since the ADBOp function is always called asynchronously, the completion routine can be used to flag call completion. Note that the function result of ADBOp indicates whether the call was successfully placed on the command queue - not whether the command has been sent to the device.
`buffer`	is a pointer to a Pascal string, which includes a length byte followed by zero to eight bytes of information. These are the zero to eight bytes that a particular register of an ADB device is capable of sending and receiving.
`commandNum`	is an integer that describes the command to be sent over the bus.

There is some confusion over the way that the completion routines are called from `_ADBOp`. This calling may be done in one of the following three ways:

- You do not wish to have a completion routine called, as in a Listen command. Pass a `NIL` pointer to `_ADBOp`.

- You wish to call the routine already in use by the system for that address (as installed by _SetADBInfo). Call _GetADBInfo before calling _ADBOp, and pass the routine pointer returned by _GetADBInfo to _ADBOp.
- You wish to provide your own completion routine and data area for the _ADBOp call. Note that the ADBOp call is always called asynchronously. In this case, simply pass your own pointers to the _ADBOp call.

The following Pascal code demonstrates a method to synchronously call ADBOp. This routine is useful for Talk commands, where the driver needs to wait for the device to return data. CallADBOp accepts the buffer and commandNum parameters and sets up a short word variable "done" as a flag variable. Initially, the flag is set to zero. CallADBOp calls ADBOp passing a pointer to the flag and to the completion routine, "CompRoutine", in addition to the buffer and commandNum parameters. The completion routine simply changes the value of the flag to -1. After calling ADBOp, the CallADBOp function enters a while loop waiting for the flag "done" to change to some non-zero value.

```
PROCEDURE SetA2;
    INLINE $34BC,$FFFF;    { MOVE.W   #$FFFF,(A2)                                  }
                           { A2 points to the variable - done - our compl flag.}
                           { Upon entry, the flag is set to zero.  Set value    }
                           { to non-zero, -1 used here, to indicate completion }

PROCEDURE CompRoutine;     { Sample ADBOp completion routine                       }

BEGIN
    SetA2;                 { Set 2 byte area pointed to by A2 to non-zero       }
END;

FUNCTION CallADBOp(buffer: Ptr; cmdNum: INTEGER): OSErr;
{ Modified version of the ADBOp function which takes the same arguments as    }
{ ADBOp except for completion routine ProcPtr.  Calls ADBOp asynchronously    }
{ then waits until the completion routine modifies "done" parameter.          }

VAR
    done:  INTEGER;
    temp:  LONGINT;
    err:   OSErr;

BEGIN
    done := 0;
    err := ADBOp(@done, @CompRoutine, @buffer, cmdNum);
    IF err = noErr THEN   { request successfully queued                         }
        REPEAT
            {Delay(2, temp);} { uncomment this line as noted below              }
            { For some time critical operations, the use of Delay procedure     }
            { has proven useful with Talk commands towards allowing the         }
            { device to complete the command.                                   }
        UNTIL (done <> 0);
    CallADBOp := err;      { 0    command entered into command queue.           }
                           { -1   command queue full, unsuccessful completion. }
END;
```

The following is the same example in C.

```
void SetA2(void)
    = {0x34BC,0xFFFF;      { MOVE.W   #$FFFF,(A2)                                  }
                           { A2 points to the variable - done - our compl flag.}
                           { Upon entry, the flag is set to zero.  Set value    }
                           { to non-zero, -1 used here, to indicate completion }
```

```
void CompRoutine(void)    { Sample ADBOp completion routine                    }

{
   SetA2;                 { Set 2 byte area pointed to by A2 to non-zero       }
}

OSErr CallADBOp(Ptr buffer, short cmdNum)
{ Modified version of the ADBOp function which takes the same arguments as    }
{ ADBOp except for completion routine ProcPtr.  Calls ADBOp asynchronously    }
{ then waits until the completion routine modifies "done" parameter.          }

{
    short  done;
    long   temp;
    OSErr  err;

    done = 0;
    err = ADBOp(&done, CompRoutine, buffer, cmdNum);
    if (err == noErr)     { request successfully queued                       }
    {
        do
            {Delay(2, temp);} { uncomment this line as noted below            }
            { For some time critical operations, the use of Delay procedure   }
            { has proven useful with Talk commands towards allowing the       }
            { device to complete the command.                                 }
        while (done != 0);
    }
    return (err);          { 0    command entered into command queue.         }
                           { -1   command queue full, unsuccessful completion. }
}
```

Remember, there should rarely be a reason to call _ADBOp. Most cases are handled by the system's polling and service request mechanism. In the cases where it is necessary to call _ADBOp, it should not be done in a polling fashion, but as a mechanism of telling the device something (i.e., change modes, or in the case of our extended keyboard, turn on or off an LED).

Question: How can I make my Macintosh II or IIx power up automatically after a power outage?

Answer: The Macintosh II and IIx power can be turned on via the keyboard through the Apple Desktop Bus port (ADB) since the reset key is wired to pin two of the ADB connector. When you press this key, it pulls pin two to ground and initiates a power-on sequence. You can emulate this feature with a momentary switch connected to the ADB port. Note that the switch on the back panel of a Macintosh IIcx and later Macintosh II models, can be locked in the On position to automatically restart after a power outage

An idea for a power-on circuit would be to have a momentary (one-shot) relay powered by the same outlet that powers the machine and have the contacts close pin two of the ADB connector. (Without having tried this, I am concerned that you may need a delay before the relay fires to give the AC time to stabilize, etc.)

Question: I'm more than a little confused about the way ADB device address conflicts are resolved at boot time.

Answer: The method used by the host to separate and identify the devices at boot time is not well documented, so I'll try to describe it with some clarity.

The host issues a Talk R3 command to an address. Let's say there are two devices at that address. Both try to respond to the command, and when they try to put the random number (the address field of register 3) on the bus, one of them should detect a collision. The one that detects the collision backs off and marks itself (internally) as unmovable.

The device that did respond successfully is then told to move to a new address (the highest free address). By definition, moving to a new address means that it now responds only to commands addressed to this new address, and it ignores commands to the original address.

The host then issues another Talk R3 command to the original address. This time the second device responds without detecting a collision. When it successfully completes a Talk R3 response, it marks itself as movable. It then is told to move to a new address.

The host again issues a Talk R3 command to the original address. Since there are no more devices at that address, the bus times out, and the host moves the last device back to the original address.

At this point, the host moves up to the next address that has a device and begins the process all over.

Generally, when having trouble separating devices on the ADB, it is because the collision detection doesn't work well. In fact, this problem is evident on Apple keyboards. The bug is that the random number returned in R3 isn't really a random number. Since the microcontrollers on the keyboards are clocked with a crystal, they tend to generate the same "random" number, so when the system attempts to separate them with a Talk R3 command, they never detect the collision.

One possible solution is to use a low-tolerance capacitor on the reset line of the microcontroller, thereby forcing the time from power on to the time reset is negated to be fairly random. In this way, the microcontroller can start a count until it receives the first Talk R3 command, and hopefully it is a different number than another device at the same address on the bus.

If you find your device shows up at all addresses, it may be because it is responding to the move address command when it should be marked as unmovable.

Finally, if the device doesn't show up at all, it may be because it is unable to respond to the Talk R3 command at boot time (i.e., not able to initialize itself and start watching the bus in time).

Further Reference:

- *Inside Macintosh*, Volume V, Apple Desktop Bus
- *Inside Macintosh*, Volume V, The Script Manager
- *The Script Manager 2.0*, Interim Chapter (DTS)
- *Macintosh Family Hardware Reference*, Chapters 11 & 19
- Technical Note #160, Key Mapping
- MacDTS Sample Code #17, TbltDrvr

Macintosh Technical Notes

#207: Styled TextEdit Changes in System 6.0

See Also: *Inside Macintosh*, TextEdit
Technical Note #31, TextEdit Bugs

Written by: Chris Derossi August 1, 1988
Revised by: Chris Derossi December 1988

Some changes were made to TextEdit in System 6.0 to provide more functionality and to make life easier for the programmer using TextEdit. This Note documents those changes and enhancements.

Changes since August 1, 1988: Corrected an error in `TEDispatchRec` in the figure on page 8.

TextEdit Changes

In order to improve the usability of styled TextEdit, some routines have been changed, and some new routines have been added. These changes exist in System Software 6.0 and later. If you intend to rely on any of these changes or new routines, it is important that you call `_SysEnvirons` first to make sure you are running under System Software 6.0 or later.

`_SysEnvirons` is documented in *Inside Macintosh*, Volume V and Technical Note #129, _SysEnvirons: System 6.0 and Beyond. To check for the styled TextEdit changes, you might do the following:

```
VAR
      theWorld:   SysEnvRec;
      anErr :     OSErr;

BEGIN
      anErr := SysEnvirons(1, theWorld);
      IF (anErr = noErr) AND (theWorld.systemVersion >= $0600) THEN …
            {System 6.0 or later}
END;
```

Changes to Existing Routines

`_TEKey` and `_TEDelete` have been changed so that backspacing to the beginning of a style no longer deletes that style. Instead, the style is saved in the `nullScrap` to be applied to subsequently typed characters. As soon as the user has backspaced past the beginning of the style, or clicked in some other area of the text, the style is removed.

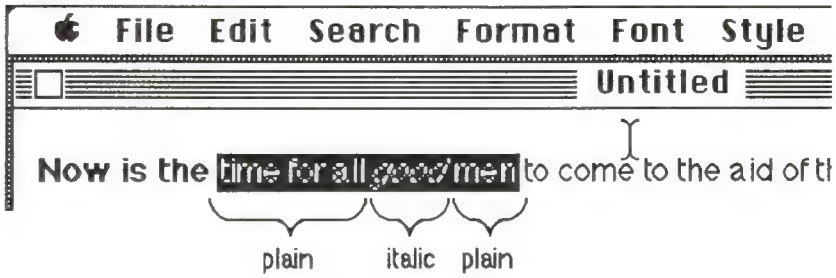
GetStylScrap now returns a handle to a valid style scrap record when called for an insertion point (selStart = selEnd). NIL is still returned when GetStylScrap is called with an old style TEHandle.

TESetStyle now accepts an additional mode, doToggle (= 32). When doToggle is specified along with doFace, TESetStyle operates as follows: If a style specified in the given TextStyle parameter exists across the entire selected range, that style is removed (turned off). Otherwise, all of the selected text is set to include that style. When a particular style is set for an entire selection range, that style is said to be continuous over the selection.

For example, given that the following text is the current selection:



then the style bold is continuous over the selection range and the italic style is not. If TESetStyle were called with a mode of doFace + doToggle and a TextStyle tsFace field of [bold], then the resulting selection would be:



On the other hand, if TESetStyle had been called with a mode of doFace + doToggle and a TextStyle tsFace field of [italic], then the selected text would have become:



New TextEdit Routines

Some new routines have been added to TextEdit, `TEContinuousStyle`, `SetStylScrap`, `TECustomHook`, and `TENumStyles`. These routines are described in detail below.

Assembly language note: The new TextEdit routines are called via the `_TEDispatch` trap. Following are the decimal selectors for the new routines:

`TEContinuousStyle`	10
`SetStylScrap`	11
`TECustomHook`	12
`TENumStyles`	13

TEContinuousStyle

```
FUNCTION TEContinuousStyle(VAR mode : Integer; VAR aStyle : TextStyle;
                           hTE : TEHandle) : Boolean;
```

`TEContinuousStyle` gives you information about the attributes of the current selection. The `mode` parameter, which takes the same values as in `TESetStyle`, specifies which attributes should be checked. When `TEContinuousStyle` returns, the `mode` parameter indicates which of the checked attributes is continuous over the selection range and the `aStyle` parameter is set to reflect the continuous attributes.

`TEContinuousStyle` returns `TRUE` if all of the attributes to be checked are continuous and `FALSE` if not. In other words, if the `mode` parameter is the same before and after the call, then `TEContinuousStyle` returns `TRUE`.

For example, `TEContinuousStyle` is useful for marking the style menu items based on the current selection.

```
mode := doFace;
IF TEContinuousStyle(mode, aStyle, myTE) THEN BEGIN
{ There is at least one face that is continuous over the
  selection. Note that it might be plain which is actually
  the absence of all styles. }
      CheckItem(styleMenu, plainItem, aStyle.tsFace = []);
      CheckItem(styleMenu, boldItem, bold IN aStyle.tsFace);
      CheckItem(styleMenu, italicItem, italic IN aStyle.tsFace);
      ...etc.
END ELSE BEGIN
{ No text face is common to the entire selection. }
      CheckItem(styleMenu, plainItem, FALSE);
      CheckItem(styleMenu, boldItem, FALSE);
      CheckItem(styleMenu, italicItem, FALSE);
      ...etc.
END;
```

This function can also be used to determine the actual values for those attributes that are continuous for the selection. Note that a field in the TextStyle record is only valid if the corresponding bit is set in the mode variable; otherwise the field contains garbage. For example, to determine the font, face, size, and color of the current selection:

```
mode := doFont + doFace + doSize + doColor;
continuous := TEContinuousStyle(mode, aStyle, myTE);
IF BitAnd(mode, doFont) <> 0 THEN
      { Font for selection = aStyle.tsFont. }
ELSE
      { More than one font in selection. };

IF BitAnd(mode, doFace) <> 0 THEN
      { aStyle.tsFace contains the text faces (or plain) that are
        common to the selection. }
ELSE
      { No text face is common to the entire selection. };

IF BitAnd(mode, doSize) <> 0 THEN
      { Size for selection = aStyle.tsSize. }
ELSE
      { More than one size in selection. };

IF BitAnd(mode, doColor) <> 0 THEN
      { Color for selection = aStyle.tsColor. }
ELSE
      { More than one color in selection. };
```

The aStyle.tsFace field is a bit tricky. When TEContinuousStyle returns a mode that contains doFace, and an aStyle.tsFace field that contains [bold, italic], it means that the selected text is all bold and all italic, but may contain other text faces as well. None of the other faces will apply to all of the selected text, or they would have been included in the tsFace field. But if the tsFace field is the empty set ([] = plain), then all of the selected text is plain.

If the current selection range is an insertion point, TEContinuousStyle returns the style information for the next character to be typed. TEContinuousStyle will always return TRUE in this case, and each field of the TextStyle record will be set if the corresponding bit in the mode parameter was set.

SetStylScrap

```
PROCEDURE SetStylScrap(rangeStart, rangeEnd : LongInt;
                       newStyles : StScrpHandle; hTE : TEHandle);
```

SetStylScrap performs the opposite function of GetStylScrap. The newStyles parameter is a handle to a style scrap record which will be applied over the given range of text. The current selection range is not changed. If newStyles is NIL or hTE is a handle to an old style TERecord, SetStylScrap does nothing.

SetStylScrap will terminate without error if it prematurely reaches the end of the range or if there are not enough scrap style elements to cover the whole range. In the latter case, the last style in the scrap record will be applied to the remainder of the range.

TENumStyles

```
FUNCTION TENumStyles(rangeStart, rangeEnd : LongInt;
                     hTE : TEHandle) : LongInt;
```

This function returns the number of style changes contained in the given range, counting one for the start of the range. Note that this does not necessarily represent the number of unique styles for the range, because some styles may be repeated. For old-style TextEdit records, this function always returns 1.

This function is useful for calculating the amount of memory that would be required for a contemplated _TECut or _TECopy. Since the style scrap record is linear in nature, with one element for each style change, you can multiply the result returned by TENumStyles by SizeOf(ScrpSTElement) and add 2 to get the amount of memory that will be needed.

TECustomHook

```
PROCEDURE TECustomHook(which : TEHook; VAR addr : ProcPtr;
                       hTE : TEHandle);
```

This procedure lets applications customize the functions of TextEdit by setting the TextEdit bottleneck routines. The which parameter specifies which bottleneck routine to replace, and is of type TEHook (described below). When TECustomHook returns, the addr parameter contains the address of the previous bottleneck routine specified by which. This is returned so that bottleneck routines can be daisy-chained.

```
TYPE
    TEHook = (intEOLHook, intDrawHook, intWidthHook, intHitTestHook);
```

The internally used fields, recalBack and recalLines now form a handle to the list of TextEdit bottleneck routines. Each TERecord has its own set of bottleneck routines to provide for maximum flexibility. The TECustomHook procedure should always be used to change the bottleneck routines instead of modifying the edit record directly.

Also, it is important to note that you should not clone a TERec. Doing so would duplicate the handle stored in recalBack and recalLines. When one of the TextEdit records was disposed, the handle stored in the copy would be invalid, and TextEdit would crash.

There are four bottleneck routines, TEEOLHook, TEWidthHook, TEDrawHook, and TEHitTestHook, described individually below. When replacing these routines, note that all registers except those specified as containing return values must be preserved. Registers A3 and A4 contain a pointer and a handle to the TextEdit record respectively. Line start positions can be obtained from the lineStarts array in the edit record.

None of these bottleneck routines are called from _TextBox.

TEEOLHook

This routine tests a given character and returns with the appropriate status flags set in the status register. The default action is to merely compare the character with `$0D` (a carriage return) and return.

On entry:	`D0`	character to compare	(byte)
	`A3`	pointer to the TextEdit record	(long)
	`A4`	handle to the TextEdit record	(long)
On exit:	`z`	flag clear if end-of-line character, set otherwise	

TEWidthHook

This routine is called any time the width of various components of a line are calculated. The appropriate font, face, and size characteristics have already been set into the current port by the time this routine is called. The default action is to call `_Char2Pixel` and return.

On entry:	`D0`	length of text to measure	(word)
	`D1`	offset into text	(word)
	`A0`	pointer to text to measure	(long)
	`A3`	pointer to the TextEdit record	(long)
	`A4`	handle to the TextEdit record	(long)
On exit:	`D1`	width of measured text	(word)

TEDrawHook

This routine is called any time the various components of a line are drawn. The appropriate font, face, and size characteristics have already been set into the current port by the time this routine is called. The default action is to call `_DrawText` and return.

On entry:	`D0`	offset into text	(word)
	`D1`	length of text to draw	(word)
	`A0`	pointer to text to draw	(long)
	`A3`	pointer to the TextEdit record	(long)
	`A4`	handle to the TextEdit record	(long)

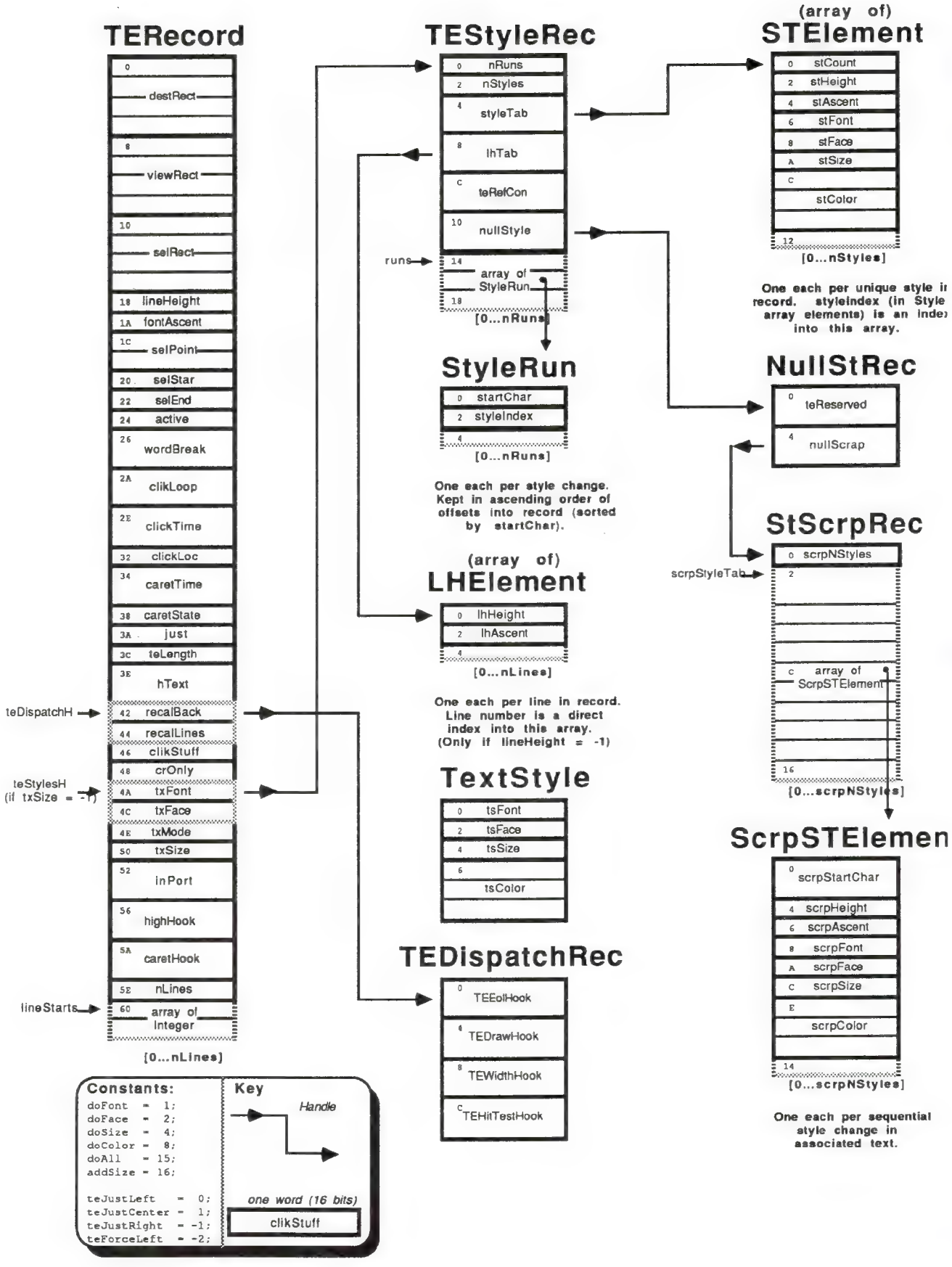
TEHitTestHook

This routine is called to determine the character position in a line given the horizontal offset, in pixels, from the beginning of a line. The default action is to call `_Pixel2Char` and return. For more information, see the description of `_Pixel2Char` in the Script Manager chapter of *Inside Macintosh*, Volume 5 and the Inside Macintosh Interim Chapter Draft, Script Manager 2.0.

On entry:	`D0`	length of text to hit test	(word)
	`D1`	pixel offset from start of text	(word)
	`A0`	pointer to start of text	(long)
	`A3`	pointer to the TextEdit record	(long)
	`A4`	handle to the TextEdit record	(long)
On exit:	`D0`	pixel width to last offset	(low word)
		Boolean = TRUE if a character offset corresponding to the pixel width was found.	(high word)
	`D1`	character offset	(word)
	`D2`	Boolean = TRUE if the pixel offset falls within the left side of the character.	(word)

TextEdit Data Structures

The illustration on the following page is a graphic representation of the TextEdit data structures. You should use this information only for debugging and so you understand what is going on. For reading or writing these data structures, the TextEdit routines should be used. This will help ensure future compatibility.


TERecord
0
destRect
8
viewRect
10
selRect
18 lineHeight
1A fontAscent
1C selPoint
20 selStar
22 selEnd
24 active
26 wordBreak
2A clikLoop
2E clickTime
32 clickLoc
34 caretTime
38 caretState
3A just
3C teLength
3E hText
42 recalBack
44 recalLines
46 clikStuff
48 crOnly
4A txFont
4C txFace
4E txMode
50 txSize
52 inPort
56 highHook
5A caretHook
5E nLines
60 array of Integer
[0...nLines]
teDispatchH
teStylesH (if txSize = -1)
lineStarts
TEStyleRec
0 nRuns
2 nStyles
4 styleTab
8 lhTab
C teRefCon
10 nullStyle
14
array of StyleRun
18
[0...nRuns]
runs
(array of) STElement
0 stCount
2 stHeight
4 stAscent
6 stFont
8 stFace
A stSize
C
stColor
12
[0...nStyles]
One each per unique style in record. styleIndex (in Style array elements) is an index into this array.
StyleRun
0 startChar
2 styleIndex
4
[0...nRuns]
One each per style change. Kept in ascending order of offsets into record (sorted by startChar).
NullStRec
0 teReserved
4 nullScrap
StScrpRec
0 scrpNStyles
2
C array of ScrpSTElement
16
[0...scrpNStyles]
scrpStyleTab
(array of) LHElement
0 lhHeight
2 lhAscent
4
[0...nLines]
One each per line in record. Line number is a direct index into this array. (Only if lineHeight = -1)
TextStyle
0 tsFont
2 tsFace
4 tsSize
6
tsColor
TEDispatchRec
0 TEEolHook
4 TEDrawHook
8 TEWidthHook
C TEHitTestHook
ScrpSTElemen
0 scrpStartChar
4 scrpHeight
6 scrpAscent
8 scrpFont
A scrpFace
C scrpSize
E
scrpColor
14
[0...scrpNStyles]
One each per sequential style change in associated text.
Constants:
doFont = 1;
doFace = 2;
doSize = 4;
doColor = 8;
doAll = 15;
addSize = 16;
teJustLeft = 0;
teJustCenter = 1;
teJustRight = -1;
teForceLeft = -2;
Key
Handle
one word (16 bits)
clikStuff

Macintosh
Technical Notes

Developer Technical Support

#208: Setting and Restoring A5

Revised by: Jim Reekes
Written by: Andrew Shebanow

June 1989
August 1988

The routines `SetupA5` and `RestoreA5` do not work properly when used with some optimizing Pascal and C compilers. Two new routines, `SetCurrentA5` and `SetA5`, are available in MPW 3.0, and they should work with any compiler.
Changes since December 1988: Removed the sample code and expanded the explanation of these two routines. The sample code in Technical Note #180 reflects these new A5 routines.

Introduction

The in-line glue routines `SetupA5` and `RestoreA5` are often used by completion routines, VBL tasks, and interrupt handlers written in C and Pascal to gain access to an application's global variables. Unfortunately, these routines play fast and loose with the stack pointer.

Newer, more sophisticated, optimizing compilers (e.g., MPW C 3.0) will often leave function parameters on the stack across multiple function calls, removing the arguments for several functions with a single instruction. This significantly reduces code size and execution time, at the expense of a small amount of additional stack usage. As a side effect, this optimization breaks the `SetupA5` and `RestoreA5` glue.

This Technical Note describes a pair of in-line glue routines which have more functionality than `SetupA5` and `RestoreA5`, without making assumptions about a compiler's stack handling. These routines are provided as a standard part of the MPW Pascal 3.0 and MPW C 3.0 packages, in the files OSUtils.p and OSUtils.h, respectively.

The Old Way

The in-line code for `SetupA5` was:

```
MOVE.L     A5,-(A7)          ; leave old A5 on stack: Danger Will Robinson!
MOVE.L     CurrentA5,A5      ; set current A5
```

The in-line code for `RestoreA5` was:

```
MOVE.L     (A7)+,A5          ; pop old A5 off stack
```

The problem is that `SetupA5` leaves the old value of `A5` on the stack, and `RestoreA5` assumes that the stack pointer is still valid. If the programmer mistakenly calls `RestoreA5` within a called subroutine, the value that is popped off the stack and stored in `A5` will be garbage. Of course, the "garbage" could be something moderately useful, like a return address.

The New, Totally Cool Way

The solution to this distressing problem is provided by two new functions in the MPW 3.0 libraries, SetCurrentA5 and SetA5. The idea behind using these two routines is to get the application's A5, pass it to some interrupt routine, which will use some globals, and then restore it to the original A5. Before you can call SetA5, you'll have to have the application's real A5. This is obtained by calling SetCurrentA5 at non-interrupt time. Here's how it should work.

An application is about to create an interrupt routine such as a VBL task, or a completion routine to be called from some asynchronous operation. To get the current A5, the application will use:

```
ourA5:= SetCurrentA5;
```

This call performs two functions. It will get the application's A5, save it in a variable, and set A5 to the application's low memory global, CurrentA5. The application will pass the result of SetCurrentA5 to the interrupt routine.

The first instruction that the interrupt routine executes is to set A5 to the application's A5. To do this the interrupt routine calls:

```
oldA5:= SetA5(ourA5);                {set A5 to the app's real A5}
                                     {perform the interrupt task}
ignoreResult:= SetA5(oldA5);         {restore A5 to the original A5}
```

The call to SetA5 performs two tasks. It will set A5 to the address specified and return the actual address in A5. You'll use the original A5 address to call SetA5 when the interrupt routine is about to exit. This action will restore A5. It's also a good idea to read Technical Note #180, which demonstrates this technique in detail.

The Interfaces

The interfaces for SetCurrentA5 and SetA5, along with their corresponding implementation as subroutine calls is:

MPW Pascal

```
FUNCTION SetCurrentA5 : LongInt;
        INLINE $2E8D, $2A78, $0904;

FUNCTION SetA5 (newA5 : LongInt) : LongInt;
        INLINE $2F4D, $0004, $2A5F;
```

MPW C

```
pascal long SetCurrentA5(void);
        { 0x2E8D, 0x2A78, 0x0904 };

pascal long SetA5 (long newA5) =
        { 0x2F4D, 0x0004, 0x2A5F };
```

Assembly Language

The following assembly-language version is for those of you who are not using a compiler capable of handling multiple word in-line functions, such as MPW C 2.0.2.

```
SetCurrentA5    MOVE.L    A5,4(A7)          ; store old A5 as function result
                MOVE.L    currentA5,A5      ; set A5 to low memory global
                RTS

SetA5           MOVE.L    (A7)+,A0          ; save return address
                MOVE.L    A5,4(A7)          ; store old A5 as function result
                MOVE.L    (A7)+,A5          ; set A5 to passed value
                JMP       (A0)
```

A Special Note

Many optimizing compilers, including the MPW C and Pascal compilers, may put the address of a global variable (i.e., a large array or record) into a register **before** your call to `SetCurrentA5` or `SetA5`. If this happens, incorrect references to your global data may be generated. To avoid this problem, you can divide your completion routine into two separate routines: one to set up and restore `A5` and one to do the actual completion work. Refer to Technical Note #180 and the following code sample for getting the application's `A5` while in an interrupt routine. The routines below will be called at interrupt time and must all be in the same code segment; otherwise, jump table references, which use `A5`, are required.

```
void DoCompletionRoutine(ParmBlkPtr pb, OSErr result)
{
        aGlobal = -1;                                   /* do some work             */
}

pascal void MyCompletionRoutine
/* This is the actual completion routine that will be called.                       */
/* There are two assembly routines not shown, GetOurA5 and                          */
/* GetPBPtr.  The application's A5 had been saved in front of                       */
/* the ParmBlk, which is pointed to in A0.  The assembly                            */
/* routine GetOurA5 obtains A5 that was stored in front of the pb                   */
/* by the application.  This technique is used in Tech Note 180.                    */
/* All of these routines must be within the same code segment.                      */
{
        long oldA5;

        oldA5 = SetA5(GetOurA5);                        /* set to our A5            */
        DoCompletionRoutine(GetPBPtr,result);           /* pbPtr is in A0           */
        (void) SetA5(oldA5);                            /* restore previous A5      */
}
```

We recommend that you switch over to the new routines as soon as possible, no matter what development system you use.

Further Reference:

- *Inside Macintosh*, Volume V-1, Compatibility Guidelines
- Technical Note #135, Getting Through CUSToms
- Technical Note #180, MultiFinder Miscellanea

#209: High Sierra & ISO 9660 CD ROM Formats

See also: AppleCD SC Developers Guide

Written by: Brian Bechtel August 1, 1988

What's Wrong with my High Sierra Disc?

Generally, if a Macintosh has problems with a High Sierra disc, it's because the disc in question doesn't really conform to the High Sierra specification. There are actually two specifications of the High Sierra format:

The Paper 28 May 1986 Working Paper for Information Processing — Volume and File Structure of Compact Read Only Optical Discs for Information Interchange (known as the "High Sierra" specification.) We'll call discs conforming to this standard "High Sierra" discs.

The Paper ISO 9660 — Volume and File Structure of CDROM for Information Interchange (known as the "ISO 9660" specification.) We'll call discs conforming to this standard "ISO" discs.

The two formats are identical in content; some fields have moved around enough to make the two formats require separate processing. Most discs pressed before 1988 are in the High Sierra format. This was the de facto standard while the international standard was being established. It appears that most discs pressed in the future will be in the ISO format.

Both standards require that you store information used to access files in two formats; least significant byte first (lsb) order (i.e. the hex number `$1234` is stored in memory as `$34 $12`) and most significant byte first (msb) order (i.e. the hex number `$1234` is stored in memory as `$12 $34`.) The 6502, 8088, 80286, and 80386 CPUs use least significant byte first order; the 68000 and 68020 use most significant byte first order.

Some of the early systems which allowed you to create High Sierra or ISO discs contained errors in the build process such that the discs didn't fully conform to the standard. In most of the cases we've seen here at Apple, one of the fields in the Primary Volume Descriptor was incorrect.

Some typical bugs:

- The path table size doesn't agree between the lsb and msb fields.
- The path table pointed at by the lsb fields doesn't contain the same data as the path table pointed at by the msb fields.

- The Standard Identifier field doesn't contain "CDROM" (for High Sierra format) or "CD001" (for ISO format.)

- The Volume descriptor version is not equal to 1.

- The File Structure Version should be 1. We allow 0, since this is a minor bug, but the correct value should be 1.

If you have a disc that you believe is in High Sierra or ISO format and the Macintosh rejects it, try the following.

1) First check to see if the files Foreign File Access, High Sierra File Access, and ISO 9660 File Access are all in your system folder. If they aren't there, you need to install them, either by using our Installer or by dragging them from the floppy that comes with the Apple CD SC drive.

2) Acquire the Validator program from AppleLink or Macintosh Developer Technical Support. Run the program, and it will tell you if it finds problems with the primary volume descriptor.

#210: The Desktop file's Outer Limits

See also: Inside Macintosh Volume 3, Page 10
Inside Macintosh Volume 4, Page 243
Technical Note #29—Resources Contained in the Desktop File
Technical Note #48—Bundles
Technical Note #141
—Maximum Number of Resources in a File

Written by: Cameron Birse August 1, 1988

There is a limit to the number of applications/files that the Finder can "see" on a single volume. This limitation is imposed by the Desktop file. The Desktop file is a resource file that the Finder uses to keep track of information about files and applications, including Finder file comments (Get Info comments), and how these files and applications relate to each other.

Because the Desktop file is a resource file, the maximum number of resources it may contain is currently 2727 (refer to Technical Note #141). To illustrate this limitation of the Desktop file, here are some example applications and how their entries currently affect the Desktop file.

- The Finder puts a single resource into the Desktop file (the Finder is not on the disk).
- MacWrite puts 10 resources into the Desktop file.
- MacPaint puts 9 resources into the Desktop file.
- MacDraw puts 8 resources into the Desktop file.
- MacWrite and MacPaint together put 20 resources into the Desktop file.
- MacWrite and MacDraw together put 19 resources into the Desktop file.
- A generic application (no BNDL resource) or a file without any Finder file comments does not put any resources into the Desktop file.
- Finder file comments put a single resource into the Desktop file.

Note: Both the maximum number of resources in a file, including the Desktop file, as well as the number of resources the above examples put into the Desktop file could change in the future.

As you can see, it is difficult to accurately predict how many applications/files will fit on any single volume. Clearly, the more information an application or file carries with it, the larger its "entry" in the Desktop file. This is normally not a problem, but with the advent of very large capacity media, it is possible to reach this limitation by creating a single volume with too many applications/files.

Macintosh Technical Notes

#211: Palette Manager Changes in System 6.0.2

See also: *Inside Macintosh*: The Palette Manager

Written by: Guillermo Ortiz October 1, 1988

This Technical Note describes the changes and enhancements to the Palette Manager in System Software 6.0.2 and future versions.

Application Palette

Applications now have the ability to define a default palette for the system to use when it needs to define the color environment (i.e., when it creates a color window without an associated palette or displays a dialog box).

The application palette feature is especially cool in cases where a color application uses old-style dialogs and alerts because without an application palette, the system will use the default palette to define the color environment. Since the system uses the default palette, the color environment may change (will change in 16-color mode) to cause some "cosmic" colors to appear in the active window. Defining a default application palette with two colors, black and white, solves this problem.

If the system needs a palette to define a color environment, it looks in the resource fork of the application for the `'pltt' ID = 0` resource and uses the palette which it contains. If the system cannot find this resource in the application's resource fork, it will use its own default palette (resource `'pltt' ID = 0` in the System file) if present, or, if necessary, it will use the Palette Manager's built-in palette.

Once an application has set its color environment (by calling `_InitMenus`, or `_InitPalettes` in weird instances when there are no menus) it can find the default palette by calling `GetPalette ( WindowPtr (-1) )` or change the default palette by calling `SetPalette ( WindowPtr (-1), newDefPltt, true )`. Note that the initialization of the Palette Manager with a call to `_InitMenus` is contrary to the way *Inside Macintosh*, Volume V, The Palette Manager documents it.

One Palette, Many Ports

You can now associate one palette with many `CGrafPort` and `CWindow` records, thus simplifying the use of a single palette with multiple ports and windows; System Software 6.0 and earlier require copies of the palette to use it with different windows.

Although this ability to associate one palette with multiple ports and windows will allow the use of calls like _PmForeColor and _PmBackColor, calling _ActivatePalette with an off-screen port does nothing, and as a result, calling it with an off-screen port will associate the palette with the port but will not cause any change in the color environment.

One important implication of this feature is that DisposeWindow (_DisposWindow) will not dispose of the associated palette automatically since it may be allocated to other ports or windows. The only exception to this behavior is when an application has used _GetNewCWindow to create the window, there is a 'pltt' resource with the same ID as the window, and the application has not called _GetPalette for the window.

Color Updates

System version 6.0.2 also introduces a new call, _NSetPalette, which complements _SetPalette. _NSetPalette has the same functionality as _SetPalette, but the CUpdates parameter has been modified from a Boolean to an Integer as follows:

```
PROCEDURE NSetPalette (dstWindow: WindowPtr; srcPalette: PaletteHandle;
                       nCUpdates: INTEGER);
INLINE $AA95;
```

_NSetPalette changes the palette associated with dstWindow to srcPalette. It also records whether the window wants to receive updates as a result of a change to its color environment. If you want dstWindow to be updated whenever its color environment changes, set nCUpdates to pmAllUpdates. If you are only interested in updates when dstWindow is the active window, set nCUpdates to pmFgUpdates. If you are only interested in updates when dstWindow is not the active window, set nCUpdates to pmBkUpdates.

```
{ NSetPalette Update Constants }

pmNoUpdates =           $8000;          {no updates}
pmBkUpdates =           $A000;          {background updates only}
pmFgUpdates =           $C000;          {foreground updates only}
pmAllUpdates =          $E000;          {all updates}
```

_SetPalette retains its syntax and function:

```
PROCEDURE SetPalette (dstWindow: WindowPtr; srcPalette: PaletteHandle;
                      CUpdates: Boolean);
INLINE $AA95;
```

Note: The trap words for _NSetPalette and _SetPalette are identical.

CopyPalette

```
PROCEDURE CopyPalette (srcPalette, dstPalette: PaletteHandle;
                       srcEntry,dstEntry,dstLength: INTEGER);
INLINE $AAA1;
```

_CopyPalette is a utility procedure that copies dstLength entries from the source palette into the destination palette; the copy begins at srcEntry and dstEntry, respectively. _CopyPalette will resize the destination palette when the number of entries after the copy is greater than the original.

_CopyPalette does not call _ActivatePalette, so the application is free to do a number of palette changes without causing a series of intermediate changes to the color environment; the application should call _ActivatePalette after completing all palette changes.

If either of the palette handles are NIL, _CopyPalette does nothing.

Macintosh
Technical Notes

Developer Technical Support

#212: The Joy of Being 32-Bit Clean

Revised by: Mensch — August 1991
Written by: Andrew Shebanow — October 1988

What to do (and what **not** to do) to make your programs run under A/UX and 32 bit versions of the Macintosh System Software.
Changes since June 1989: Updated the information for system 7.0.
Changes since October 1988: Added information on writing 32-bit clean CDEFs, and updated A/UX information to reflect the capabilities of A/UX 1.1.

Introduction

Some programs available today will not run in a 32-bit world. Previously the Macintosh OS ran only in a 24-bit world, where the hardware ignores the high byte of all memory addresses (including pointers and handles). Under A/UX and system software 7.0, programs can run in a 32-bit world, where the entire address is significant. This Technical Note presents guidelines which you should follow to make your program work in the 32-bit world.

Note: Much of the information presented here has already been discussed in one or more of the documents referenced at the end of this Note, but it is being repeated here because of the importance of the subject matter.

Keep in mind that the rules presented here are not graven in stone. Although you may find it necessary to break some of these rules to achieve specific functionality in your program, it is important to remember that in doing so, your application may break in 32 bit environments. Keeping your program compatible is your responsibility as well as Apple's.

General Rules

The following are some general rules that you should follow to make your program more robust:

- Always code defensively. Check the error code after you make a call to the Toolbox. Make sure your handles and pointers are not `NIL`. Do not assume that calls will always succeed. See Technical Note #117 for more information.
- Use `_SysEnvirons` and `_Gestalt` to determine your system's configuration. When checking for the processor type, make allowances for newer Motorola processors like the 68030 & 68040. See Technical Note #129 for more information on `_SysEnvirons` and `_Gestalt`.
- Do not try to check to see if MultiFinder is active (you cannot tell anyway); your application should work properly with and without MultiFinder.

- Do not make assumptions about the maximum size of a piece of memory or a resource. Do not assume that the maximum distance between two objects in memory is less than 2^{32} bytes. Do not store less than 32 bits for the size of an object (e.g., PICTs) in your data structures unless you create them.
- Call `_NGetTrapAddress` for any traps you use that are not available under A/UX (i.e., SCSI Manager traps). For a complete list of traps available under A/UX, see *A/UX Toolbox: Macintosh ROM Interface*.
- Always use the latest version of your development system and documentation. Even though you do everything "correctly," earlier versions may have bugs or inaccuracies that could break your program.

To summarize: don't make any assumptions, even if those assumptions are currently true. The future **will** change.

Hardware & CPUs

- Do not assume that you are running in the processor's supervisor mode. Do not use `TRAP` instructions or exception vectors that are reserved for future use by Apple or Motorola. See the Compatibility Guidelines chapter of *Inside Macintosh*, Volume V-1 for more information.
- Never try to bypass existing interfaces to hardware devices. Direct hardware access is not available under A/UX. Use the Serial Driver to talk to the SCC. Use the File Manager to manipulate disks. Use the SCSI Manager to talk to your non-disk devices like scanners and printers. Use QuickDraw to draw to your screen.
- Do not use timing loops. Different CPUs execute them at different speeds.

Memory Manager

Memory Manager abuse is the leading cause of death under 32-Bit operation. Here are some crucial points to remember:

- Do not set bits in master pointers directly. Use Memory Manager traps (e.g., `_HLock`, `_HGetState`, and `_HSetState`) instead.
- Do not use fake handles under any circumstances. See Technical Note #117 for more information on `Handle` etiquette.
- When you compare master pointers, use `_StripAddress` to convert them to the correct format. See the OS Utilities chapter of *Inside Macintosh*, Volume V and Technical Note #213 for more information.
- Do not make assumptions about the contents of Memory Manager data structures, including master pointers and zone headers. These structures have changed under A/UX, and they will change again in the future.

Resource Manager

Here are some guidelines for using the Resource Manager:

- Avoid opening resource files read-only unless the resource file is on an AppleShare volume. If another application (including DAs and other types of code) opens (or already has open) the resource file for writing, you could end up with a corrupted

resource map. If you do open a resource file read-only, you should load the resources you need into memory immediately.

- Do not set resource attribute bits directly. Use the supplied `_GetResAttrs` and `_SetResAttrs` traps.
- Do not make assumptions about the contents of Resource Manager data structures and, especially, the resource map. Do not try to walk the resource map.
- Follow the special procedures outlined below whenever loading and using any resource that contains executable code.

WDEFs and CDEFs

In earlier versions of the System Software, the Window Manager and the Control Manager both stored the variant code (which defines how the window or control looks) in the high byte of the `defProc` field. You should use the `_GetWVariant` and `_GetCVariant` traps to get the variant code. If writing your own WDEF or CDEF, you should use the variant parameter that is passed to you.

If you are writing your own CDEF, you have to be very careful. Prior to System 7.0, there was no way to make a CDEF fully 32-bit clean, since the `calcCRgns` message uses the high bit of the region handle as a flag. *Inside Macintosh*, Volume I-331 says to "clear the high byte (not just the high bit) of the region handle before attempting to update the region." **This is wrong.** You should clear just the high bit, or your code will not run under A/UX or future versions of the Macintosh OS.

The Control Manager in System 7.0 or later has a new mechanism for calculating control regions. The following two new messages have been defined:

```
CONST
        calcCntlRgn = 10;
        calcThumbRgn = 11;
```

Your CDEF can indicate that it handles these new messages by setting the result of your CDEF dispatching function to non-zero. Whenever the Control Manager used to call your CDEF with the `calcCRgns` message, it will now do the following:

```
        IF we are in 32-bit mode
                rgnHandle := Call CDEF with message calcCntlRgn OR calcThumbRgn, as appropriate
                If CDEF returned 0 then call CDEF with old calcCRgns.
        ELSE
                call CDEF with old calcCRgns method, just like the good old days
```

Old CDEFs will continue to run in 24-bit mode, but they may not correctly draw in all instances (however, they will not crash).

If your program has custom CDEFs, you should update them to support the new messages (along with the old ones) as soon as possible. Supporting the new messages will allow them to work correctly today, and in the future.

Low-Memory Globals

A/UX does not support all of the low-memory globals, and future systems may provide even less support. Unfortunately, there are some things you just cannot do on a Macintosh without using

low-memory globals, so it is currently impossible to avoid them entirely. Here are some guidelines:

- Avoid writing to or reading from low-memory globals unless absolutely necessary.
- Do not use low-memory globals that are labeled private, reserved for future use, or that are undocumented.
- Do not use low-memory globals when there is a trap or library routine which accomplishes the same task. For example, A/UX does not have an event queue that you can access in low memory, but it does support all of the Event Manager traps for accessing the Event Queue (e.g., `_GetNextEvent`, `_WaitNextEvent`, `_EventAvail`).

Trap Patches

Patching traps is one of the easiest ways to break your program. It is very difficult to write a trap patch that does not make incorrect assumptions about the way things work. Many current applications patch traps unnecessarily. If a trap does not work the way you want, implement your own code instead of trying to patch the required functionality into the trap. Here are a few guidelines to follow if you absolutely must patch a trap:

- Do not assume that `A5` is valid when you call the patch.
- Do not bypass the Trap Dispatcher to call traps directly. The performance gains are small, and there may be serious side effects.
- Do not use the Memory Manager if the trap that you are patching is not listed in "Appendix A: Routines That Move Or Purge Memory" of *Inside Macintosh X–Ref*.

Make sure that any patch you do write is not a tail patch. A tail patch is a patch which looks at the results returned by the original patch and modifies them to suit its own purposes. If you call the original trap routine with a `JSR` instead of a `JMP`, you have created a tail patch.

You need to avoid tail patches because many of Apple's System Software patches check the return address on the stack to see who called them. If you write a tail patch, you defeat these checks and may cause things to break in strange and less than wonderful ways.

Application loaded code type (stand alone) resources

If your program loads any resources that contain executable code, you should not assume that the pointer you get from the handle is 32-bit clean. Before calling any routines in any code resources you load you should call `_StripAddress` first to insure that the routine gets a 32-bit clean PC. Not all existing applications that load external code resources perform this operation (nor is the system gauranteed to do so before calling an INIT), so if you are writing a code resource of this type you should be sure to always call `_StripAddress` to clean the PC as documented in tech note #228 at the main entry point of all external routines and INITs. This particularly important if you use any PC relative addressing modes or are calling `_SwapMMUMode`.

Further Reference:

- *Inside Macintosh*, Volume V-1, Compatibility Guidelines
- *Inside Macintosh*, Volume VI, Compatibility Guidelines
- *Inside Macintosh X–Ref*
- Technical Note #2, Compatibility Guidelines
- Technical Note #117, Compatibility: Why & How
- Technical Note #129, _SysEnvirons: System 6.0 and Beyond
- Technical Note #213, _StripAddress: The Untold Story
- Technical Note #228, Use Care When Swapping MMU Mode



#213: _StripAddress: The Untold Story

Revised by: Paul Snively & Andrew Shebanow
Written by: Andrew Shebanow

August 1990
October 1988

Inside Macintosh, Volume V, The OS Utilities, incorrectly documents the `_StripAddress` trap; this Technical Note correctly documents the trap and gives guidelines for its use.

Changes since April 1990: Added a discussion of why the `_StripAddress` trap should be used under certain circumstances when patching traps.

Don't Believe Everything You Read

The `_StripAddress` trap is described in the OS Utilities chapter of *Inside Macintosh*, Volume V as:

```
FUNCTION StripAddress(theAddress: LONGINT): LONGINT;
```

> *If the system is running in 24-bit mode, `_StripAddress` is identical in function to the global variable `Lo3Bytes`: it returns the value of the low-order three bytes of the address passed in `theAddress`. If the system is in 32-bit mode, however, `_StripAddress` simply passes back the address unchanged.*

This description is incorrect, and it is located in the wrong chapter of *Inside Macintosh* (it should be in the Memory Manager chapter). The `_StripAddress` trap now takes a `Ptr` as a parameter and returns a `Ptr`:

```
FUNCTION StripAddress(theAddress: Ptr): Ptr;
```

The effect of `_StripAddress` on the passed address depends on the native (bootup) mode the operating system uses, and **not** on the current mode. The following chart shows the results of `_StripAddress` in all of its configurations:

Operating System	24-Bit MMUMode	32-Bit MMUMode
System Software < 7.0	masked with `Lo3Bytes`	masked with `Lo3Bytes`
System 7.0 (24-Bit Bootup)	masked with `Lo3Bytes`	masked with `Lo3Bytes`
System 7.0 (32-Bit Bootup)	*(never in 24-bit mode)*	address unchanged
A/UX 1.1.1	*(never in 24-bit mode)*	address unchanged
A/UX 2.0 (24-Bit Login)	masked with `Lo3Bytes`	masked with `Lo3Bytes`
A/UX 2.0 (32-Bit Login)	*(never in 24-bit mode)*	address unchanged

Should I Call _StripAddress Each Time I Use An Address?

In a word, **no**. Twenty-four-bit addresses are not inherently dangerous—they only cause problems when you access them in 32-bit mode, so unless you are switching modes with the `_SwapMMUMode` trap, you rarely need to call `_StripAddress` inside a "typical" application or driver. Calling `_StripAddress` is **unnecessary** in the following three cases:

Dereferencing A Pointer or A Handle

You don't need to call `_StripAddress` before dereferencing a pointer or a handle unless you are switching the CPU into 32-bit mode with the `_SwapMMUMode` trap. After all, if this is a full-time 32-bit machine, the pointer is always a valid 32-bit address, and if it is a 24-bit machine, your addresses are valid 24-bit addresses unless you switch the machine into 32-bit mode yourself.

Comparing Pointers And Handles For Equality

As long as you don't futz with the high bits of pointers and handles yourself (which you cannot do safely if you want to run in 32-bit mode in any case), you should be able to compare pointers and handles for equality without doing a `_StripAddress`, since the high byte always contains the same "garbage" when you are in 24-bit mode that it did when the pointer or handle was created. There is an exception to this rule, which is discussed in the "Comparing Dereferenced Handles" section later in this Note.

Performing Address Arithmetic

You do not need to call `_StripAddress` before performing address arithmetic, since adding or subtracting two 24-bit addresses always results in the correct 24-bit address, regardless of the high byte values. Random high byte values can cause ordered comparisons on the results of pointer arithmetic to fail, since underflow or overflow could occur, so you should never test the result of address arithmetic against a value (only against `NIL` or 0).

Okay, So When Do I Need To Call _StripAddress?

Ordered Address Comparison

If you need to sort by address or do any other kind of ordered address comparison, you need to call `_StripAddress` on each address before doing any ordered comparisons (>, <, >=, <=). Remember, even though the CPU only uses the lower 24 bits in 24-bit mode, it still uses all 32 bits when performing arithmetic operations.

Comparing Dereferenced Handles (Master Pointers)

If you want to perform any type of comparison on dereferenced handles, you need to call `_StripAddress` on the value first, since the Memory Manager stores its flags in the high byte of the Master Pointer on 24-bit machines, and these flags can change at any time. Of course, this implies that you need to call `_StripAddress` before comparing two pointers for equality if you could be comparing against a dereferenced handle.

On Handles And Pointers Before Dereferencing in 32-Bit Mode

If you are going to switch the machine into 32-bit mode yourself, you need to call `_StripAddress` on all 24-bit pointers and handles that you access while in 32-bit mode. Of course, you had better **not** call `_StripAddress` on a 32-bit address (for example, a 32-bit NuBus™ slot address could generate some very nice fireworks if you strip off its high byte with `_StripAddress` and then try to access the "improved" address). For example, the 32-Bit QuickDraw routine `_GetPixBaseAddr` returns a 32-bit address. Refer to Technical Note #277, 32-Bit QuickDraw: Version 1.2 Features, for more details about 32-Bit QuickDraw and 32-bit addressing.

Program Counter in 32-Bit Mode

This issue is described in depth in Technical Note #228, Use Care When Swapping MMU Mode, so this Note won't go into depth here. Basically, if you are going to switch into 32-bit mode within a code resource, you need to make sure that your Program Counter (`PC`) contains a valid 32-bit address **before** you switch modes.

One not-so-obvious example of this case is a trap patch that lives in a block in the heap. If you pass the address of the block to the `_SetTrapAddress` trap without first calling `_StripAddress` on it and the patched trap is later called in 32-bit mode, bad things happen. Specifically, the high byte of the address contains Memory Manager information, so when the patched trap is called, the `PC` points to never-never land, making results extremely unpredictable. So if you are going to patch a trap using the address of a heap block, call `_StripAddress` on it first.

Filenames Passed To _OpenResFile And _OpenRFPerm

This issue is described in depth in Technical Note #232, Strip With `_OpenResFile` and `_OpenRFPerm`. A patch to these traps calls `_RecoverHandle` on the string passed to these routines, which can cause crashes if `_StripAddress` is not called.

Whaaahh! _StripAddress Is Too Slow! How Can I Make It Faster?

If you follow the guidelines discussed in this Note, speed shouldn't be an issue, since you are calling `_StripAddress` rarely, if at all. Just for the sake of argument, though, let's say that you do call `_StripAddress` inside of a time-critical loop inside an interrupt routine. The solution to this problem is to cache the mask that `_StripAddress` uses to do its work. Here's how:

MPW C:

```
long gStripAddressMask;        /* our cached global mask variable */

/* you can use this macro to do a quick _StripAddress equivalent */
#define QuickStrip(ptr) ((ptr) & gStripAddressMask)

main()
{
  /* do all of the usual initialization */
  /* cache _StripAddress result */
  gStripAddressMask = 0xffffffff;
  gStripAddressMask = (long) StripAddress((Ptr) gStripAddressMask);
  /* have your program do something useful here… */
}
```

MPW Pascal:

```
PROGRAM StripTease;
VAR
  gStripAddressMask : LONGINT;        { our cached global mask variable }

{ you can use this function to do a quick _StripAddress equivalent }
FUNCTION QuickStrip(thePtr : Ptr) : Ptr;
BEGIN
  QuickStrip := Ptr(BAND(LONGINT(thePtr),gStripAddressMask));
END;

BEGIN
  { do all of the usual initialization }
  { cache _StripAddress result }
  gStripAddressMask := $FFFFFFFF;
  gStripAddressMask := LONGINT(StripAddress(Ptr(gStripAddressMask)));
  { have your program do something useful here… }
END.
```

This technique avoids the overhead of the Trap Dispatcher, works on present and future machines, and it should be fast enough for any application—just call the `QuickStrip` routine (macro to us C weenies) in place of `_StripAddress`.

Further Reference:

- *Inside Macintosh,* Volume V, The OS Utilities
- Technical Note #212, The Joy Of Being 32-Bit Clean
- Technical Note #228, Use Care When Swapping MMU Mode
- Technical Note #232, Strip With `_OpenResFile` And `_OpenRFPerm`
- Technical Note #275, 32-Bit QuickDraw: Version 1.2 Features

NuBus is a trademark of Texas Instruments.

#214: New Resource Manager Calls

See also: *Inside Macintosh*: The Resource Manager
The File Manager
Technical Note #101: _CreateResFile and the Poor Man's Search Path

Written by: Andrew Shebanow October 1, 1988

This Technical Note describes two new Resource Manager calls that make opening and creating resource files much easier.

MPW 3.0 supplies glue routines for two new Resource Manager calls which provide new, easier ways of opening and creating resource files.

The important thing about these two calls is that they allow you to pass a directory ID instead of a working directory `refNum`. This means that you can create a resource file without worrying about the Poor Man's Search Path (PMSP). If you try to create a file using `_CreateResFile`, and there is already a resource file with the same name in the System Folder (or in any other folder that is on the PMSP's list), the `_CreateResFile` call will not work because the Resource Manager thinks the resource file already exists. The all new `HCreateResFile` glue does not use the PMSP if you specify a non-zero directory ID.

```
FUNCTION HOpenResFile(vRefNum: INTEGER;
                      dirID: LONGINT;
                      fileName: Str255;
                      permission: SignedByte): INTEGER;
```

The `HOpenResFile` routine opens an existing resource file in the directory specified by `vRefNum` and `dirID` and it returns the `refNum` of the resource file. If the `refNum` equals –1, you should call `_ResError` to check for errors. This routine also lets you open a resource file without creating a working directory.

```
PROCEDURE HCreateResFile(vRefNum: INTEGER;
                         dirID: LONGINT;
                         fileName: Str255);
```

The `HCreateResFile` routine creates a new resource file with name `fileName` in the directory specified by `vRefNum` and `dirID`. You should call `_ResError` to check for errors.

#215: "New" cdev Messages

See also: *Inside Macintosh*, Volume V, The Control Panel

Written by: Mark Bennett October 1, 1988

This Technical Note describes some previously undocumented messages that the Control Panel can send to a Control Panel device (cdev).

The Control Panel will send messages to a Control Panel device (cdev) in response to the user selecting the Undo, Cut, Copy, Paste and Clear items of the Edit menu. It will also send a message if the cdev contains a `'CURS' = -4064` resource. The following is a list of the previously undocumented messages, descriptions, and values:

Message	Description	Value
`undoDev`	Undo selected	9
`cutDev`	Cut selected	10
`copyDev`	Copy selected	11
`pasteDev`	Paste selected	12
`clearDev`	Clear selected	13
`cursorDev`	Cursor resource	14

The Control panel only sends the `undoDev`, `cutDev`, `copyDev`, `pasteDev`, and `clearDev` messages to a cdev as a result of the Desk Manager sending an edit message to it when an application calls `SystemEdit (_SysEdit)`. Since the call to `SystemEdit (_SysEdit)` is triggered by a mouse-down event in the menu bar, the messages to the cdev will be sent only as a result of the user selecting the Edit menu item with the mouse and not by pressing the Command-key equivalent.

Typically, you will call `_DlgCut`, `_DlgCopy`, `_DlgPaste` or `_DlgDelete` upon receipt of the `cutDev`, `copyDev`, `pasteDev`, or `clearDev` message, passing the `DialogPtr` that has been passed to the cdev to the call.

To respond to Command-key equivalents of the Edit menu commands, you must check for the specific characters and modifier keys themselves, even though this is never localized. Once you determine the character to be a Command-key equivalent, you must alter the `what` field of the event record that has been passed to the cdev to be a `nullEvent` to prevent the Dialog Manager from inserting the character into the `editText` item of the cdev. To alter the event record, you should treat the event record parameter which is passed to the cdev as a reference. In Pascal, this means declaring the interface to the cdev as follows:

```
FUNCTION MyCdev(message, item, numItems, CPanelID: INTEGER
                VAR theEvent: EventRecord; (* the 'NEW' way *)
                cdevStorage: Handle;
                CPDialog: DialogPtr) : Handle;
```

In C, you would do the following:

```
Handle MyCdev(message, item, numItems, CPanelID
              theEvent, cdevStorage, CPDialog)
short message, item, numItems, CPanelID;
EventRecord *theEvent;              /* the 'NEW' way */
Handle cdevStorage;
DialogPtr CPDialog;
```

In assembly language, it means you do not make your own copy of the event record, so you are probably already set up to change the value of the `what` field of the event record.

If the cdev contains a `'CURS' = -4064` resource, the Control Panel will send it a `cursorDev` message whenever the cursor is over the cdev part of the Control Panel's window instead of setting the cursor to the light cross. The cdev can then set and use its own cursor. The Control Panel will handle the cursor elsewhere on the screen. The Control Panel does not examine the contents of the `'CURS' = -4064` resource.

#216: AppleShare 1.1 and 2.0 Limits

See also: *Inside Macintosh*: The AppleTalk Manager
The File Manager
Technical Note #186: PBLock/UnlockRange

Written by: Mark Bennett October 1, 1988

This Technical Note describes some machine-dependent limits of current versions of AppleShare and AppleShare servers.

The following chart lists some current AppleShare limits which are based upon the chosen server platform and memory configuration. The limits which otherwise might be present on a workstation are still in effect and are not affected by the workstation being logged into an AppleShare server. These limits will change in the future.

Description of Limit	Server machine is: Macintosh Plus, SE, or II with one Mb	Server machine is: Macintosh II with more than one Mb
Number of Users	25	50
Number of Locked Ranges	1000	2000
Number of Open Files	80	160
Number of Volumes	16	16

Macintosh
Technical Notes



Developer Technical Support

#217: Where Have My Font Icons Gone?

Revised by: Pete "Luke" Alexander April 1991
Written by: Pete "Luke" Alexander December 1988

This Technical Note discusses why you should not link directly from your font files to the font icons provided by LaserWriter driver 5.2 and later.
Changes since December 1988: Added some useful tips and described the method required to bundle an icon to your font file.

Introduction

This Technical Note discusses why you should not directly link your font files to the font icons provided by the LaserWriter driver 5.2, and what you can do to still use these font icons. This Note applies only to PostScript® downloadable font files for use with PostScript printers, not font files which have been created by the Font/DA Mover.

In the past, it was possible to directly link your PostScript downloadable font files to the font icons provided by the LaserWriter driver. This is no longer possible, because the `'FREF'` and `'ICN#'` resources do not match in LaserWriter driver 5.2 and later.

Oh, But Why Change Things Now?

Apple engineering decided that they did not want developers using the font icons directly from the driver, they wanted the option to change them in the future. Due to time constraints, the original font icons were not removed from the LaserWriter driver 5.2, but the `'FREF'` and `'ICN#'` IDs were changed, thereby preventing developers from linking directly to the icons. Engineering has removed the font icons from LaserWriter driver 6.0 and later.

If you want to use Apple's font icons from LaserWriter driver v5.2, you should copy them into the resource fork of your font file. Apple expects all developers to bundle their own icons with their font file.

What Do I Need To Do?

You need to do the same things you did, when you bundled your icon to your application. Although your font file is just a document, if you want your "custom icon" to appear when your application is not present, the font file needs to contain the following resources: `'BNDL'`, `'FREF'`, and `'ICN#'`. *Inside Macintosh*, Volume III, The Finder Interface and Technical Note #48, Bundles, explain the details of bundling icons with your font files.

So, How Do I Do This From Within An Application?

To enable you application to add icons to the font file when it is created, you need to install a copy of the 'BNDL', 'FREF', and 'ICN#' resources (used by the font file to bundle it's icon to the file), into the resource fork of the application. Make sure that they do not conflict with the same resources used by the application to bundle its icon(s).

When creating a "new" font file, copy the 'BNDL', 'FREF', and 'ICN#' resources from the resource fork of the application, into the resource fork of the font file. You will also need to set the bundle bit of the font file, otherwise Finder will not realize that the file contains an icon. The font file will now be able to show it's icons, whether the application is present or not.

Conclusion

With LaserWriter driver v5.2 and later, the 'FREF' and 'ICN#' resources do not match, therefore you will need to bundle your custom font file icon within it's resource fork. If you provide the 'BNDL', 'FREF', and 'ICN#' resources in the resource fork of your font file, Finder will display your custom icon for the font file, whether the application is present or not.

Further Reference:

- *Inside Macintosh*, Volume III, The Finder Interface
- Technical Note #48, Bundles

PostScript is a registered trademark of Adobe Systems, Incorporated.

#218: New High-Level File Manager Calls

See also: *Inside Macintosh*, Volumes IV & V, The File Manager
Technical Note #140, Why PHBSetVol is Dangerous

Written by: Andrew Shebanow December 1988

This Technical Note describes some new high-level File Manager calls that make dealing with the Hierarchical File System (HFS) easier.

When the Hierarchical File System (HFS) was first introduced, a large number of low-level File Manager calls were documented. Unfortunately, higher-level equivalents to these calls were not present until now. The glue for these routines is built into MPW 3.0. They are provided as a convenience for those of you who hate filling in parameter blocks.

The new routines can be divided into two groups: routines which provide new high-level capabilities for working with HFS volumes and routines which work like the existing high-level calls, but let you use a directory ID as a parameter.

New High-Level Routines

```
FUNCTION AllocContig(refNum: INTEGER; VAR count: LONGINT) : OSErr;
```

This routine works like the `Allocate` routine, except that it allocates contiguous space on disk for the specified file. If the required space cannot be allocated, a `dskFullErr` will be returned.

```
FUNCTION HCreate(vRefNum: INTEGER; dirID: LONGINT; fileName: Str255;
                 creator: OSType; fileType: OSType) : OSErr;
```

This routine creates a new file like `PBHCreate`, and then sets the type and creator of the file for you.

```
FUNCTION DirCreate(vRefNum: INTEGER; parentDirID: LONGINT;
                   dirName: Str255; VAR createdDirID: LONGINT) : OSErr;
```

This routine creates a directory for you. It returns the new directory ID in the `createdDirID` parameter.

```
FUNCTION CatMove(vRefNum: INTEGER; dirID: LONGINT; oldName: Str255;
                 newDirID: LONGINT; newName: Str255) : OSErr;
```

This routine moves a file or directory to a new location on the same volume.

```
FUNCTION OpenWD(vRefNum: INTEGER; dirID: LONGINT;
                procID: LONGINT; VAR wdRefNum: INTEGER) : OSErr;
```

This routine opens a new working directory. It returns the working directory's reference number in the `wdRefNum` parameter.

```
FUNCTION CloseWD(wdRefNum: INTEGER) : OSErr;
```

This routine closes a working directory.

```
FUNCTION GetWDInfo(wdRefNum: INTEGER; VAR vRefNum: INTEGER;
                   VAR dirID: LONGINT; VAR procID: LONGINT) : OSErr;
```

This routine returns information about an existing working directory.

Hierarchical Versions of Existing High-Level Routines

```
FUNCTION HGetVol(volName: StringPtr; VAR vRefNum: INTEGER;
                 VAR dirID: LONGINT) : OSErr;

FUNCTION HSetVol(volName: StringPtr; vRefNum: INTEGER; dirID: LONGINT) : OSErr;
```

Note: `_HSetVol` can be hazardous to your health (see Technical Note #140, Why PBHSetVol is Dangerous).

```
FUNCTION HGetFInfo(vRefNum: INTEGER; dirID: LONGINT;
                   fileName: Str255; VAR fndrInfo: FInfo) : OSErr;

FUNCTION HSetFInfo(vRefNum: INTEGER; dirID: LONGINT;
                   fileName: Str255; fndrInfo: FInfo) : OSErr;

FUNCTION HOpen(vRefNum: INTEGER; dirID: LONGINT; fileName: Str255;
               permission: SignedByte; VAR refNum: INTEGER) : OSErr;

FUNCTION HOpenRF(vRefNum: INTEGER; dirID: LONGINT; fileName: Str255;
                 permission: SignedByte; VAR refNum: INTEGER) : OSErr;

FUNCTION HDelete(vRefNum: INTEGER; dirID: LONGINT; fileName: Str255): OSErr;

FUNCTION HRename(vRefNum: INTEGER; dirID: LONGINT;
                 oldName: Str255; newName: Str255) : OSErr;

FUNCTION HSetFLock(vRefNum: INTEGER; dirID: LONGINT; fileName: Str255) : OSErr;

FUNCTION HRstFLock(vRefNum: INTEGER; dirID: LONGINT; fileName: Str255) : OSErr;
```

#219: New Memory Manager Glue Routines

See also: *Inside Macintosh*, Volume I, The Memory Manager

Written by: Andrew Shebanow December 1988

This Technical Note describes some new Memory Manager routines which make life a little easier for C and Pascal programmers.

MPW 3.0 includes some new glue routines that allow you to allocate pre-zeroed handles and pointers and to allocate memory (zeroed or otherwise) in the system heap. These capabilities have always been available to assembly language programmers, but these routines make it possible for C and Pascal programmers to achieve the same results.

Here are the definitions for the new routines:

```
FUNCTION NewHandleSys(byteCount: Size): Handle;
```

Allocate a new handle in the system heap.

```
FUNCTION NewHandleClear(byteCount: Size): Handle;
```

Allocate a new handle, and fill the allocated memory with zeros.

```
FUNCTION NewHandleSysClear(byteCount: Size): Handle;
```

Allocate a new handle in the system heap, and fill the allocated memory with zeros.

```
FUNCTION NewPtrSys(byteCount: Size): Ptr;
```

Allocate a new pointer in the system heap.

```
FUNCTION NewPtrClear(byteCount: Size): Ptr;
```

Allocate a new pointer, and fill the allocated memory with zeros.

```
FUNCTION NewPtrSysClear(byteCount: Size): Ptr;
```

Allocate a new pointer in the system heap, and fill the allocated memory with zeros.

#220: Segment Loader Limitations

See also: *Inside Macintosh*, Volume II, The Segment Loader

Written by: Andrew Shebanow December 1988

This Technical Note discusses the jump table limitations of the Segment Loader and suggests some ways to work around these limitations to minimize the problem. These limitations are most evident to developers using MacApp and other object-oriented environments.

As application sizes increase, more and more developers are hitting a little known limit in the Macintosh run-time architecture: the Segment Loader's limit on the number of jump table entries.

The Segment Loader reads the jump table from `'CODE'` resource 0. This resource has the following format:

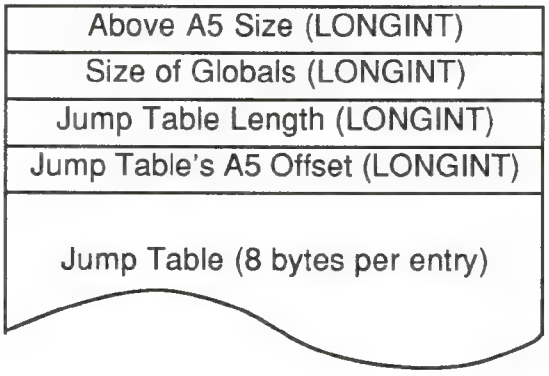


Figure 1–Jump Table in 'CODE' Resource 0

Since the 68000 uses signed 16-bit offsets for `A5`-relative instructions, the maximum offset that can be specified is 32767. Since each jump table entry takes up 8 bytes, there can be 4096 jump table entries, but we need to subtract the 16 bytes used by the jump table header, so the actual limit on jump table entries is **4094**.

This limit may seem rather small, but you have to remember that jump table entries are only needed if a function is called from outside of its own segment (a.k.a. an intersegment function call). For routines that are local to a segment (intrasegment function calls), the linker can call the routine without going through the jump table.

If you are developing a large program, and you think you might be in danger of running out of jump table space, here are some guidelines you can use when designing and organizing your code:

- Segment your source code based on your run-time call chain, rather than on a simple file-by-file basis.
- Keep utility routines in the same segment as the routines which call them, rather than keeping them in a separate "Utilities" segment.
- Try to keep your segments close to the maximum size. The fewer segments you have, the fewer intersegment references you will have.

The 4094 routine limit becomes a much more significant problem when you start working with MacApp and Object Pascal. If you compile your MacApp program with debugging turned on, you will get a jump table entry for **every** method in your program. All of a sudden, an application of medium size can have jump table problems. Working around this problem is extremely difficult, since the only options available to you are:

- Giving up the MacApp debugger facilities, and using your favorite low-level debugger instead.
- Reorganizing your classes, and reducing the number of methods in your program to a reasonable level. This means that you may have to make major source code changes, and you end up with a less "object-oriented" program than you started out with.

Both of these options are extremely distasteful, but the final version of MacApp 2.0 will alleviate the problem by letting you compile your programs with both debugging and optimization turned on. The optimizer will drastically reduce the number of jump table entries, so the problem should only occur with very large programs.

Macintosh
Technical Notes



Developer Technical Support

#221: NuBus Interrupt Latency (I Was a Teenage DMA Junkie)

Revised by: Cameron Birse October 1989
Written by: Cameron Birse, Mark Baumwell, & Rich Collyer December 1988

This Technical Note discusses NuBus™ interrupt latency, and why, contrary to popular belief, the Macintosh is **not** a real-time machine.
Changes since December 1988: Changed sample code to defer cursor rendering to a deferred task rather than a "pseudo-VBL" task.

The Macintosh is **not** a real-time machine. The Macintosh does **not** support DMA. There are many variables in the Macintosh that make it impossible to deterministically figure out exactly when things are going to happen. Despite these facts, there are those who must push the envelope. For these courageous adventurers, we provide the following information in the hope that it speeds your journey.

According to empirical evidence gathered by Apple engineering, typical NuBus to Macintosh transaction times fall in the 800 nanosecond to 1 microsecond range. Although the NuBus specification points to faster accesses, you should consider these times realistic since there is always some overhead. Synchronizing the NuBus and Macintosh clocks, for example, can cost a NuBus cycle.

One technique that can help optimize NuBus transfers is implementing bus locking. The bus can be locked for a small set of transactions (we recommend a maximum of four transfers), then unlocked for rearbitration. In order to allow fairness, it is important to lock the bus for as short a time as possible.

All processor interrupts and slot interrupts may be held off for various amounts of time by different parts of the system, so you must never count on instant interrupt response. To help deal with these delays, you should consider your data rate and include ample buffering on your card for your data. The following are just a few of the many system variables which affect interrupt latency:

- Floppy disk accesses turn off interrupts for "significant" (read milliseconds) amounts of time. For instance, some disk accesses (i.e., block reads) can disable interrupts for as much as 15 milliseconds. Inserting a blank floppy disk turns off interrupts for up to 25 milliseconds.
- Formatting a floppy disk turns off interrupts for up to 300 milliseconds.
- LocalTalk accesses can disable interrupts for up to 22 milliseconds.
- Assuming your interrupt handler is going to want to access your card immediately, there is also the arbitration for mastership of the bus, which could be in use at the time, and in the worst case, lock the bus, keeping you from accessing your card.
- All slot interrupts, including slot VBL interrupts, hold off other slot interrupts. This means another card's interrupt routine (installed via `_SIntInstall`) or a slot VBL interrupt routine (installed via `_SlotVInstall`) runs to completion

with interrupts of the slot level and below disabled. VBL tasks may be of varying length, since applications, as well as drivers, can and do, install VBL tasks.

- Cursor updating (performed during slot VBL time) time ranges from around 700 µSec - 900 µSec for one-bit to eight-bit depth. Since this is done at slot VBL time, it holds off all other slot interrupts until it is finished.

Warning: The performance figures cited in this Note are based on current Macintosh models; they are not guaranteed to remain the same in future machines.

The following code lets you defer the cursor updating routine by having it run as a deferred task. This change means that the actual cursor rendering is performed with interrupts enabled, which allows the occurrence of other interrupts. It should be noted that there is a slightly visible flickering of the cursor as a result of using this technique.

```
;*****************************************************************
;***
;***     Defer Cursor
;***     This program defers the cursor updating that normally happens
;***     during slot VBL time. Since the cursor updating can take as
;***     long as 900µSec, and holds off other slot interrupts, it is
;***     handy to be able to defer the updating to a more civilized time.
;***     This program replaces the normal jCrsrTask with a routine that
;***     installs the real jCrsrTask routine as a deferred task.
;***
;***     Build commands:
;***
;***     asm DeferCrsr.a -lo DeferCrsr.a.lst -l
;***     link DeferCrsr.a.o -o DeferCrsr
;***
;*****************************************************************

                                STRING ASIS
                                PRINT  OFF
                                INCLUDE 'Traps.a'
                                INCLUDE 'SysEqu.a'
                                PRINT  ON

;******************************** Entry ********************************

Entry   MAIN

        bra.s           Entry2

;****************************** MyDefTask ******************************

TaskBegin
MyDefTask

                DC.L    0       ;qLink (handled by OS)
                DC.W    0       ;qType (equ 7, find this value in MPW AIncludes)
                DC.W    0       ;dtFlags (reserved, don't mess with 'em)
                DC.L    0       ;dtAddr (pointer to actual routine to be performed)
                DC.L    0       ;dtParm (optional parameter, this example doesn't use it)
                DC.L    0       ;dtReserved (should be zero, DC.L 0 takes care of that)

SysCrsrTask
                DC.L    0
```

```
***************************** MyjCrsrTask ****************************

MyjCrsrTask
        movem.l         a0/d0,-(sp)
        lea             MyDefTask,a0                ;point to our deferred task element
        move.l          SysCrsrTask,dtAddr(a0)      ;set up pointer to routine
        move.w          #dtQType,dtType(a0)         ;set queue type
        _DTInstall                                  ;install the task
        movem.l         (sp)+,a0/d0
        rts
TaskEnd

******************************** Entry2 ******************************

TaskSize        EQU             TaskEnd-TaskBegin

Entry2
        move.l          #TaskSize,d0            ;TaskSize = Deferred task element, room for
                                                ; a pointer (to original jCrsrTask), and
                                                ;our jCrsrTask
        _NewPtr         ,SYS,CLEAR              ;make a block in the system heap
        bne.s           Abort                   ;no room at the Inn, head for the manger
        move.l          a0,a2                   ;got a good pointer, keep a copy
        move.l          a0,a1                   ;a0 = source, a1 = destination for
                                                ; BlockMove
        lea             MyDEFTask,a0            ;copy the task, etc. into the system heap
        move.w          #TaskSize,d0
        _BlockMove

        lea             dtQElSize(a2),a0        ;move original jCrsrTask pointer into our
        move.l          jCrsrTask,(a0)          ; pointer holder
        lea             dtQElSize+4(a2),a0      ;replace jCrsrTask pointer with a pointer
        move.l          a0,jCrsrTask            ; to our jCrsrTask

abort   rts                                     ;all's well that ends…

        END
```

- Note, as an aside, that while using MacsBug, interrupts are disabled.

In summary, you cannot depend on real-time performance when transferring data between NuBus and the Macintosh. It is important to provide sufficient buffering on the card to allow for the variance in interrupt latency. Driver calls can be used to determine the amount of data available to be transferred, and transfers can be made on a periodic basis.

Remember too, since the entire system is so heavily interrupt-driven, it is very unfriendly for anyone to disable interrupts and take over the machine for long periods of time. Doing so almost always results in a sluggish user interface, something which is usually not well received by the user.

Further Reference:

- *Inside Macintosh*, Volume V, The Device Manager
- *Inside Macintosh*, Volume V, The Vertical Retrace Manager
- *Macintosh Family Hardware Reference*
- *Designing Cards and Drivers for the Macintosh II and Macintosh SE*

NuBus is a trademark of Texas Instruments

Macintosh
Technical Notes

Developer Technical Support

#222: Custom Menu Flashing Bug

Written by: Dennis Hescox February 1989

Selected menu items in a custom `'MDEF'` resource do not flash correctly due to a bug in the Menu Manager. This Technical Note describes the problem and explains how to make your `'MDEF'` flash correctly.

Not on the Menu

There is a known bug in the Menu Manager's interface to custom `'MDEF'` resources which causes an item to flash incorrectly if its corresponding menu record contains no text. If there is no text in the the chosen menu record, the Menu Manager neglects to call the menu definition procedure multiple times with `mChooseMsg`, and the corresponding item does not flash. If there is text in the menu record, the Menu Manager calls the menu definition procedure multiple times to flash the menu item.

No Substitutions Allowed

Until a corrected version of the Menu Manager is released, you can make your custom menu items flash by adding a dummy text menu item in the menu record corresponding to each item in your custom `'MDEF'`.

For Example...

When using a Macintosh with color, notice that items in the Finder's Color menu (other than the first one) do not flash. Edit the `'MENU'` resource (id=16) in a copy of the Finder. After menu item "x" add menu items "b," "c," "d," "e," "f," "g," and "h" (or other legends of your own choosing) to correspond to the additional seven Color menu items. After rebooting with this edited Finder, selections from the Finder's Color menu should now flash correctly.

Further Reference:

- *Inside Macintosh*, Volume I-339, The Menu Manager

Macintosh
Technical Notes



Developer Technical Support

#223: Assembly Language Use of _InitGraf with MPW

Written by: Dennis Hescox February 1989

The Macintosh Programmer's Workshop (MPW) requires assembly-language programmers to allocate their own QuickDraw global variables rather than use the default record as indicated in *Inside Macintosh.*

In the Beginning...

Early Macintosh assembly-language development systems automatically allocated a set of QuickDraw global variables (a QuickDraw record) on the application's stack. The assembly-language note in *Inside Macintosh*, I-163 indicates that the programmer should use the following code to specify that automatically allocated record to `_InitGraf`:

```
PEA         -4(A5)
_InitGraf
```

Here and Now

Despite the note in *Inside Macintosh*, MPW does not automatically allocate a set of QuickDraw global variables for the programmer; it is the responsibility of the programmer to allocate this record for QuickDraw's use.

An Example...

Here is an example of creating a QuickDraw global variables template taken from Sample.a, V1.01, which is distributed with MPW 3.0 and on Developer Technical Support's sample source code disk:

```
QDGlobals     RECORD  0,DECREMENT
GrafPort      DS.L    1
White         DS.B    8
Black         DS.B    8
Gray          DS.B    8
LtGray        DS.B    8
DkGray        DS.B    8
Arrow         DS.B    cursRec
ScreenBits    DS.B    BitMap
RandSeed      DS.L    1
              ORG     -GrafSize
              ENDR
```

Here is an example, again from Sample.a, of how to use the above template.

First we use the template to allocate an occurrence of our record:

```
QD              DS              QDGlobals               ; QuickDraw's globals
```

Next we set up for and make our call:

```
                PEA             QD.GrafPort
                _InitGraf
```

The declaration of `QD` with a `DS` creates an occurrence of our record in the application's global variable space and the assembler implicitly provides the reference to `A5` as the base register .

Truth or Consequences

If you use the MPW assembler and also use the code specified by the note in *Inside Macintosh*, you are telling QuickDraw (by calling `_InitGraf`) to use an inappropriate area of memory as global variables space. QuickDraw will happily initialize this area of memory to the correct values for its use, but in doing so, it will also be blasting information that is probably important to some other part of the system.

Further Reference:

- *Inside Macintosh*, Volume I-135, QuickDraw

Macintosh
Technical Notes

Developer Technical Support

#224: Opening AppleTalk

Written by: Mark Bennett February 1989

This Technical Note describes the most effective, safe, and compatible way to open the AppleTalk drivers, .MPP and .ATP.

The process of opening the AppleTalk drivers, .MPP and .ATP, can be greatly simplified. The AppleTalk Manager chapters of *Inside Macintosh* describe the calls `MPPOpen` and `ATPLoad` for use by high-level languages. They also describe the process of examining low-memory globals `SPConfig` and `PortBUse` before calling `_Open` for assembly language use of AppleTalk.

Starting with the 128K ROM, the .MPP driver already has all the code built in for checking the low-memory globals `SPConfig` and `PortBUse` before trying to complete the `_Open` call. Furthermore, the .MPP driver will automatically open the .ATP driver as part of its opening process. Therefore, since all of the required checks are made inside the driver itself, we recommend that a simple `_Open` call be made to the .MPP driver when you need to use AppleTalk. In a high-level language like Pascal, this call would look like the following:

```
result := OpenDriver('.MPP', refnum);
```

In C:

```
result = OpenDriver("\p.MPP", &refnum);
```

And in assembly language:

```
openAT      SUB.W       #ioQElSize,SP       ; Make space for paramblock on stack since
                                            ; _Open is always synchronous. Using .W is
                                            ; slightly more efficient and is safe since
                                            ; ioQElSize is small.
            MOVE.L      SP,A0               ; Point A0 to paramblock.
            LEA         mppName,A1          ; Point A1 to driver name.
            MOVE.L      A1,ioFileName(A0)   ; Put pointer to name in paramblock.
            CLR.B       ioPermssn(A0)       ; Clear so won't look like OpenDeskAcc.
            _Open
            MOVE.W      ioRefNum(A0),D1     ; You might want this later. Who knows?
            ADD.W       #ioQElSize,SP       ; Reclaim space on stack.
            RTS                             ; D0 contains result code.

mppName     DC.B        4
            DC.B        '.MPP'
```

By using just the simple `_Open` call to the .MPP driver, you can ensure that your code will be compatible with future versions of AppleTalk that might not make use of low-memory globals.

Further Reference:

- *Inside Macintosh*, Volumes II-261, IV-229, & V-507, The AppleTalk Manager

---



#225: Using RegisterName

Written by: Mark Bennett February 1989

The verify flag indicator byte (`verifyFlag`) of the AppleTalk `RegisterName` function should always be set `TRUE` in published code.

The AppleTalk chapter of *Inside Macintosh*, Volume II-322, in describing the `RegisterName` function, states:

> *"If verifyFlag is TRUE, RegisterName checks on the network to see if the name is already in use, and returns a result code of nbpDuplicate if so."*

Note that `verifyFlag` should **always** be `TRUE` in published code. The design of the Name Binding Protocol (NBP) requires that an entity name be unique. The way `RegisterName` ensures this uniqueness is by broadcasting a lookup request for the registered name. If any entity responds, then `RegisterName` knows that the name would not be unique and returns an error. Some developers, in anticipating the time delay involved in broadcasting a lookup request and waiting for a response, have opted to set `verifyFlag` to `FALSE`, not realizing the potential danger in doing so.

Apple provides `verifyFlag` for experimental or developmental purposes, such as narrowing down a problem in registering a name on a network. Always make sure that code which ships has `verifyFlag` set to `TRUE`.

Further Reference:

- *Inside Macintosh*, Volume II-261, The AppleTalk Manager

Macintosh
Technical Notes



Developer Technical Support

#226: Moving Your Cat

Revised by: John Harvey
Written by: John Harvey

February 1991
February 1989

This Technical Note clarifies the documentation in *Inside Macintosh* for `_PBCatMove` and provides a demonstration on how to use it.

Changes since February 1989: Added a discussion of using `NIL` for the destination name pointer, which is the simplest way to use `_CatMove`, and revised the sample code to use the high-level File Manager calls. Thanks to Tim Dierks of Apple Developer Technical Support U.K. for pointing out the problems with the previous version of this Note.

The documentation for `_PBCatMove` is not particularly clear; *Inside Macintosh*, Volume IV-157, states:

> *"The name and directory ID of the directory to which the file or directory is to be moved are specified by ioNewName and ioNewDirID."*

To understand this explanation, imagine a situation where your program maintains a work file in the default directory. Since this work file can get really big, if the user asks to save it to the same disk as the default directory, you use `_PBCatMove` to move the file instead of copying it. After the user dismisses `SFPutFile` by clicking on the Save button, you have the situation illustrated in Figure 1.

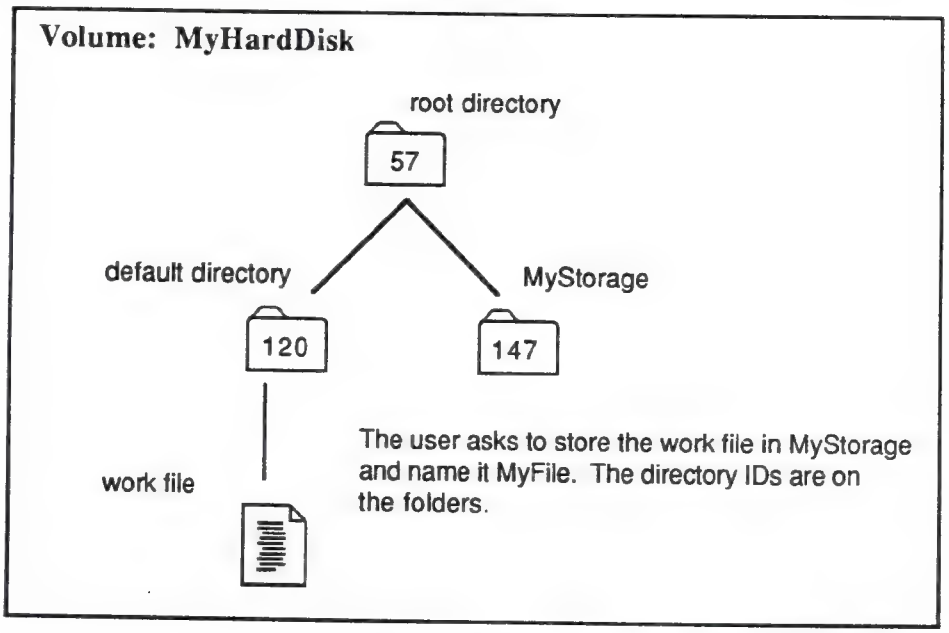


Figure 1–Directory Structure After SFPutFile

Following the directions in *Inside Macintosh*, you put the directory ID (147) in `ioNewDirID` and a pointer to the directory name (the string "MyStorage") in `ioNewName`, but if you do this, you get a `fnfErr` (-43), since the File Manager looks for the directory inside of itself.

To understand this situation, consider how the File Manager accesses files or directories. It uses four different methods: full pathname, volume reference number and partial pathname, directory ID and partial pathname, and working directory reference number and partial pathname. Most File Manager calls let the programmer choose any of these methods, but `_PBCatMove` seems to only allow the third method, directory ID and partial pathname, to access a file or directory. Figure 2 illustrates why a call to `_PBCatMove` fails if you use 147 as the directory ID.

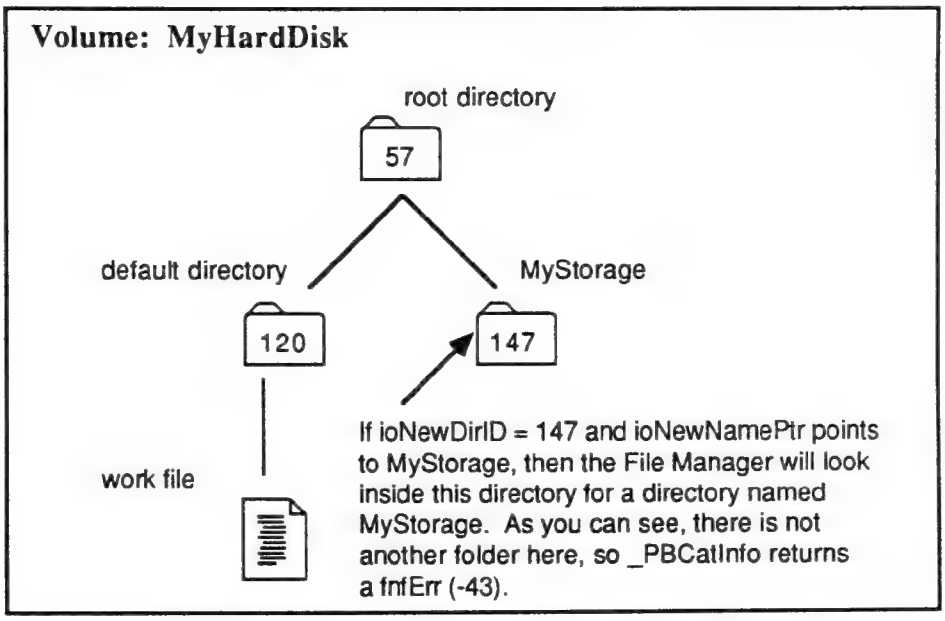


Figure 2–_PBCatMove Failure

One approach to solving this problem is to identify the directory that contains the destination directory (i.e., the destination's parent directory) by putting its directory ID in the `ioNewDirID` and identifying the destination directory by placing a pointer to its name in `ioNewName`. Taking that approach means that `_PBCatMove` uses the `ioNewDirID` to find the folder that contains the destination folder, and `ioNewName` to find the actual destination folder. Figure 3 illustrates this process.

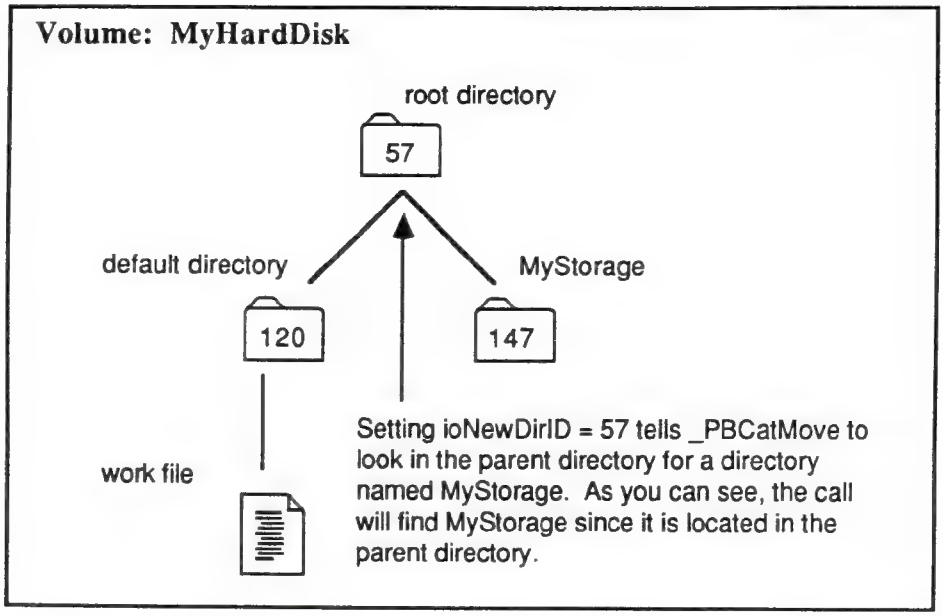


Figure 3–Parent and Destination Directories

However, there are two drawbacks to this approach, one minor and one major. The minor one is that you need to dig for the parent directory ID and allocate memory for the source directory's name. This is not a big deal; however, sometimes it can be a pain. The major drawback is that supplying a parent directory ID for the destination is **impossible** when the destination is the root. The root is the parent of all directories and is its own parent (i.e., try calling `_GetCatInfo` with the `ioDirID` set to the root directory ID—the returned `ioDrParID` is the same value as the one to which you set `ioDirID`).

Better Living Through NIL

However, as is usually the case with the File Manager, you can perform wonders with the discreet use of `NIL` pointers. If you set `ioNewName` to `NIL`, you can then set `ioNewDirID` to the directory ID of the actual destination directory, and all proceeds as desired. This method is simpler then having to come up with a parent directory ID and the name of the destination directory. In addition to working in all situations, it also saves space in your code since you do not need to provide space for the name of the destination.

The line from *Inside Macintosh*, Volume IV-157, can be rephrased to say, "The name of the directory to which the file or directory is to be moved is specified by `ioNewName`. If a valid directory name is provided for `ioNewName`, then the destination directory's **parent** directory is specified by `ioNewDirID`. However, you may specify `NIL` for `ioNewName`, in which case `ioNewDirID` should be set to the directory ID of the destination directory."

Following are examples in Pascal and C which demonstrate this simpler method of using `_PBCatMove`. To make life even easier, these examples have been revised to use the high-level File Manager calls introduced in Technical Note #218, New High-Level File Manager Calls.

Most Of The Time That Is

Using the high-level calls provides the opportunity to point out a minor issue with MPW Pascal and the high-level `CatMove` call. As documented in this Note, you want to set `ioNewName` to `NIL`; unfortunately, MPW Pascal does not let you pass `NIL` as the destination name when you call the high-level `CatMove`. Even more unfortunate, if you pass `''` (an empty string) to `CatMove`, it returns -37 (bad file or volume name). There is one thing you can do to work around this problem, and the Pascal example in this Note uses it. Things still work fine if you pass `':'` as the name of the destination directory to `CatMove`, and this makes sense since `':'` is a partial pathname, which, in this case, simply means that `_CatMove` should place the file in the directory specified by `ioNewDirID`. This latter method corresponds to the directory ID and partial pathname method described earlier in this Note.

It is important to understand that this is only an issue when using the high-level `CatMove` in Pascal. It is perfectly fine to set `ioNewName` to `NIL` if you are using the low-level `PBCatMove` in Pascal—the compiler accepts it and everything works correctly.

Pascal

```
PROCEDURE MoveTheCat( sourceWD: INTEGER; destinationWD: INTEGER;
                      fname: Str255);
{------------------------------------------------------------------------------
|    MoveTheCat
|      Routine to move a file from one directory to another via CatMove. Note
|      that the directories MUST be on the same volume.  This routine expects
|      the caller to have checked this already.
|    Note on Error Handling:
|      This example uses Signal to handle any errors.  Please see Tech Note
|      #88 for more information on Signal.
|      Parameters:
|        sourceWD:  Where the file is now.  This is a Working Directory
|                   that standard file returned.
|        destinationWD:  Where we want to move the file.  Again a working
|                        directory returned by standard file.
|        fname:   The name of the file we are moving
------------------------------------------------------------------------------}
VAR
    Volume            :INTEGER;
    sourceDirID       :LONGINT;
    destinationDirID  :LONGINT;
    procID            :LONGINT;

BEGIN

  {**
      First we use the High-Level GetWDInfo call to convert our working
      directories to directory IDs.
  **}
    Signal( GetWDInfo( sourceWD,Volume, sourceDirID, procID) );
    Signal( GetWDInfo(destinationWD,Volume,destinationDirID,procID) );

  {**
      Now move that file. Note use of ':' since compiler won't let us use NIL and
      '' does the wrong thing.
  **}
    Signal( CatMove(sourceVol,sourceDirID, fname,destinationDirID, ':') )
END;
```

C

```
void MoveTheCat( short sourceWD, short destinationWD, Str255 fname )
{

    short    Volume;
    long     sourceDirID;
    long     destinationDirID;
    long     procID;

    Signal( GetWDInfo(sourceWD,&Volume,&sourceDirID,&procID) );
    Signal( GetWDInfo(destinationWD,&Volume,&destinationDirID,&procID) );

    /**
       Note we can use nil for the destination name in C
    **/
    Signal( CatMove( Volume, sourceDirID, fname, destinationDirID, nil ) );
}
```

Further Reference:

- *Inside Macintosh*, Volume IV-87, The File Manager
- Technical Note #88, Signals
- Technical Note #218, New High-Level File Manager Calls

Macintosh
Technical Notes

Developer Technical Support

#227: Toolbox Karma

Written by: Ed Tecot February 1989

This Technical Note discusses Macintosh Toolbox compatibility and what you can do to help the Macintosh continue evolving in the future.

It is getting increasingly difficult to make additions to the Macintosh Toolbox. The single greatest obstacle today is compatibility. Often, engineering is prevented from doing something in an elegant manner because it would break some applications. This usually leaves three choices:

1. Break the application. Engineering does not normally choose this course of action.
2. Don't support the feature. This is rarely a good choice. It is bad for the user, and it limits developers.
3. Implement the feature in a less-than-optimal way. This is the choice most often taken. Examples are the auxiliary window list, faking desk accessories in MultiFinder to force clipboard conversion, and the ever unpopular menu bar definition procedure (MBDF).

Engineering doesn't like making additions in this way, since it clutters the architecture and makes Macintosh programming even more difficult.

Rules, Rules, Rules

You're probably thinking, "But I followed the rules." You're right. You've followed the stated guidelines in *Inside Macintosh* and the Macintosh Technical Notes. You've done nothing explicitly wrong.

However, you can do **more** than just follow the rules. Consider what effect your design decisions have on the Macintosh community. Understand that by taking advantage of a documented feature, you may be preventing the Macintosh from growing in the future. If you follow some of the following guidelines, you can give Apple some flexibility in changing rules that are no longer appropriate. These guidelines are just a sample, and hopefully you can extrapolate more from this list.

Traps Are Here to Stay

The trap interface is the ultimate Macintosh standard. Even when data structures change, the traps always work. Use them to their fullest. Don't directly manipulate data structures when a trap call will do, don't use `_HandToHand` to duplicate a handle if there is an explicit trap call available (e.g., `_TENew`), and don't patch traps. If a trap does not work the way you want, implement your own code instead of trying to patch the required functionality into the trap. If you absolutely must patch a trap, don't make assumptions about registers (e.g., `A5`) or modify the stack. See Technical Note #212, The Joy of Being 32-Bit Clean, for more information on the evils of patching traps.

Data Structures Are Subject to Change

Engineering won't haphazardly change them, but by using the traps, you give them the flexibility to make these changes. If everyone had been using `_SetWRefCon` and `_GetWRefCon`, the auxiliary window list might not have been necessary. Of course, if everyone agrees to use these traps **and** leave the auxiliary window list alone, maybe they can fix this one in the future.

Write Robust Definition Procedures

All of the definition functions, WDEF, CDEF, MDEF, etc. have room for growth. Do not stunt this growth by making unnecessary assumptions. If you do not understand the message, don't do anything. If a parameter is documented as unused, don't use it; it may be used in the future. These same rules apply to anything which might be called from ROM, such as drivers, user procedures, and filter procedures. Treat the MBDF as undocumented. It has changed considerably in the past and will continue to do so.

Use Globals With Caution

Globals often have their meaning changed or their format altered. Use the trap interfaces when available (e.g., `_TickCount` instead of `Ticks`). Try to avoid using them at all if possible.

Your Future is Apple's Future

As a developer you play a key role in shaping the future of the Macintosh. By going beyond the guidelines in *Inside Macintosh* and the Macintosh Technical Notes and considering the effects of your design decisions on the whole Macintosh community, you allow the Macintosh to grow and change while still maintaining compatibility. We won't break your applications, we can fully support features you desire, and we can implement these features in the best possible way for us, for you, and for the users. By going that extra step, you help us make programming the Macintosh simpler and ensure the best possible future for your products as well as ours.

Further Reference:

- *Inside Macintosh*, Volume V, Compatibility Guidelines
- Technical Note #2, Compatibility Guidelines
- Technical Note #117, Compatibility: Why & How
- Technical Note #212, The Joy of Being 32-Bit Clean

Macintosh
Technical Notes

Developer Technical Support

#228: Use Care When Swapping MMU Mode

Revised by: Andrew Shebanow
Written by: Cameron Birse

February 1990
April 1989

This Technical Note describes how to avoid crashing when swapping into 32-bit mode on a Macintosh II. Thanks to Jim Berry and Dan Weston for pointing this out.
Changes since April 1989: Added a reference to Technical Note #213, _StripAddress: The Untold Story.

There is a condition where calling `_SwapMMUMode` to switch the Macintosh II into 32-bit mode can cause the system to crash. This condition happens in code which is loaded into memory from a resource, or is placed in memory that was allocated by the Memory Manager and is subsequently executed by using the master pointer as the address for a `JSR` instruction. This condition includes stand-alone, executable code resources (i.e., `'XCMD'`, `'XFCN'`, `'INIT'`, `'ADBS'`,`'FKEY'`, etc.), but does **not** apply to standard `'CODE'` resources because the Segment Loader fixes the PC.

When you load code into memory as a resource in 24-bit mode (i.e., by calling `_GetResource`), the high byte of the master pointer contains Memory Manager information. If you perform a `JSR` to the code (typically a `JSR (A0)` with the master pointer in `A0`), the entire master pointer gets translated directly into the program counter, including the high byte of Memory Manager information. As soon as you switch into 32-bit mode, the program counter effectively has garbage in the high byte, and the machine goes directly into the weeds (do not pass go, do not collect $200).

You can avoid this problem by cleaning up the program counter from within the resource code before calling `_SwapMMUMode`. The following example shows how to clean up the PC using MPW Pascal and C with inline assembly code:

MPW Pascal

```
PROCEDURE FixPC;
    INLINE  $41FA, $000A,  {   LEA        *+$000C,A0                    }
            $2008,         {   MOVE.L     A0,D0                         }
            $A055,         {   _StripAddress                            }
            $2040,         {   MOVEA.L    D0,A0                         }
            $4ED0;         {   JMP        (A0) ;jmps to next instruction }
```

MPW C

```
pascal void FixPC()
      = {0x41FA, 0x000A,   /*   LEA        *+$000C,A0                    */
         0x2008,           /*   MOVE.L     A0,D0                         */
         0xA055,           /*   _StripAddress                            */
         0x2040,           /*   MOVEA.L    D0,A0                         */
         0x4ED0};          /*   JMP        (A0) ;jmps to next instruction */
```

Further Reference:

- *Inside Macintosh*, Volume V-591, OS Utilities
- Technical Note #212, The Joy of Being 32-Bit Clean
- Technical Note #213, _StripAddress: The Untold Story

Macintosh
Technical Notes



Developer Technical Support

#229: A/UX 2.0 Compatibility Guidelines

Revised by: Kent Sandvik & B. Winston Hendrickson — February 1991
Revised by: B. Winston Hendrickson & Dave Radcliffe — June 1990
Written by: Dave Radcliffe — April 1989

This Technical Note describes details of the A/UX 2.0 implementation of which developers should be aware, so that their Macintosh applications also work properly under A/UX.
Changes since April 1989: This Note formerly described A/UX 1.1 Toolbox Bugs, but has been completely rewritten to cover A/UX 2.0 compatibility.
Changes since June 1990: Changes due to A/UX 2.0.1, also added some new important issues.

Introduction

A/UX 2.0 and 2.0.1 significantly improves support for Macintosh applications from version 1.1. The major components of the 2.0 Toolbox environment include:

- System Software derived from 6.0.5
- A/UX MultiFinder version 6.9
- 32-Bit QuickDraw
- File Manager access to UNIX, HFS, and AppleShare volumes
- Sound Manager
- Serial Manager
- Notification Manager
- Slot Manager

Most MultiFinder-compatible Macintosh applications and utilities should work unmodified under A/UX 2.0. Complete details concerning the A/UX Toolbox may be found in the *A/UX Toolbox: Macintosh ROM Interface* manual which is part of the A/UX documentation set; however, there are some additional subtle aspects of the A/UX environment of which developers should be aware.

General Compatibility Issues

The following items do not form a definitive list. Rather, they are indicative of common problems Apple encountered with applications while testing them under A/UX 2.0.

Do Not Access Hardware Directly

UNIX is a protected environment where the kernel arbitrates all hardware access. Applications should use the appropriate Macintosh Toolbox manager when it is available. Hardware developers may need to write a custom A/UX device driver to provide access to special hardware. Details on A/UX device drivers, including sample driver source code, is contained in the A/UX Device Driver Kit available from APDA.

Do Not Depend Upon How Things Work

Some applications rely on internal operating system details. This reliance has always been a bad idea, but applications are particularly likely to be bitten under A/UX because the internals are different. Following are some specific cases of which to be aware:

Event Manager

Some applications depend upon being able to access the event queue directly and patch into the `_PostEvent` mechanism. Under A/UX, the event queue is in the kernel and is not accessible to applications. Use if possible existing supported trap calls, instead of trying to access the event queue.

Memory Manager

Some applications work because the Macintosh OS uses a roving allocation scheme. Such applications free memory then use that memory in subsequent traps. A/UX is much more likely to reuse that memory immediately after it is freed (i.e., before the application can use it) than the Macintosh OS.

A/UX-Specific System Files

Some system files are A/UX specific and cannot be interchanged with the Macintosh environment. These files include the System file, Finder, MultiFinder, AppleShare, MacTCP and Patch files.

Privileged Instructions

Since privileged microprocessor instructions must be emulated in software, they execute much slower under A/UX than the Macintosh OS; therefore, using privileged instructions degrades application performance. Avoid a privileged instruction where a non-privileged instruction does just as well. For example, the non-privileged instruction `MOVE <ea>,CCR` can be used in almost all cases instead of the privileged instruction `MOVE <ea>,SR`.

Personal System Folders

Under A/UX, a user has the option to use a public System Folder ("`/mac/sys/System Folder`") which may be shared and changed by all users or employ a private System Folder to allow custom configurations. When the user logs in, A/UX makes two checks for the existence of a private System Folder. First, the environment variable `TBSYSTEM` may be set to the full UNIX path of a directory to use as the System Folder. Second, if `TBSYSTEM` is not set, the user's home directory is checked for a subdirectory named "`System Folder`". If neither of these conditions are met, the public System Folder is used. Once a particular directory is selected, it becomes the "blessed" folder as long as the user remains logged in.

If an application has an installer utility, it should not place required files in the active System Folder; rather, it should use the System Folder only for configuration data. Required files should be kept with the application.

File Manager

The A/UX File Manager provides access to: UNIX® volumes, HFS volumes, MFS volumes, and AppleShare volumes. UNIX volumes may be a Berkeley 4.2 File System (UFS), a System V File System (SVFS) or a Network File System (NFS™) volume. In addition, since HFS is used as the foundation of the 2.0 File Manager, other external file systems following the Macintosh compatibility guidelines should work correctly under the A/UX environment. In fact, access to UNIX file systems is provided through a Macintosh external file system.

HFS Volumes

A/UX 2.0 allows only one HFS partition per volume; volumes with more than one such partition are not usable. In addition, HFS SCSI devices are only accessible to A/UX if they were accessible to the Macintosh OS when the system booted, and they have a valid partition map. Note that in the case of HFS CD-ROM discs, only the drive needs to be accessible, no volume needs to be in the drive.

UNIX Volumes (/)

Different physical UNIX volumes (including NFS volumes) appear to Macintosh applications as a **single** Macintosh volume named `/`. Moving files within the single logical volume `/` using `_PBCatMove` can fail with a `badMoveErr` if the move would cross physical volumes. It also means the free space reported on `/` may not be an accurate reflection of the space available for a file operation. The `_AUXDispatch` trap, documented in the *A/UX Toolbox: Macintosh ROM Interface* manual, with the `AUX_FS_FREE_SPACE` selector, can be used to accurately determine the free space on the underlying physical volume.

The `/` volume always appears as the boot volume and **always** contains the "blessed" folder.

Filenames

UNIX file systems, unlike Macintosh file systems, are **case sensitive**. Applications should be consistent in the use of case to avoid confusion. For example, the filenames "`Foobar Preferences`" and "`foobar preferences`" are two distinct filenames to UNIX. The A/UX 2.0 File Manager mediates this a bit. It first attempts a case sensitive filename lookup, and if that fails, it then tries a case insensitive lookup. The disadvantage is that not only is this slower, but also the File Manager may fail to correctly identify a single file from a group of files whose names differ only in case. Note that the `_Create` trap is an exception and only performs a **case sensitive** lookup when it checks for the existence of a file, thus an application may create "`Foo`" in the same directory which already contains "`foo`".

Not all UNIX file systems support 31-character filenames. The UFS file system on which A/UX 2.0 is based, supports 255-character filenames, but the older System V file systems used on A/UX 1.x systems only support 14-character filenames. Since these older systems are supported under A/UX 2.0, users may mount them as part of `/`. While the File Manager continues to allow use of 31-character filenames, remember that the underlying file system may truncate the name. For example, on an SVFS volume, the name `ThisNameWillBeTruncated` becomes `ThisNameWillBe`. This also means names like `ThisNameWillBeTruncated` and `ThisNameWillBeTruncatedAlso` may not be unique. You can avoid problems by ensuring that static filenames used by application code are unique in the first 14 characters.

UNIX Volume Pathnames

The slash (/) character is used by UNIX as a pathname separator. Its use is similar, but not identical, to that of the colon (:) in Macintosh file systems. This similarity can lead to ambiguity; for example, "`Font/DA Mover`" might be interpreted as "`:Font:DA Mover`". Again, the File Manager attempts to arbitrate. It accepts **both** UNIX and HFS style pathnames, and if it is trying to parse a pathname according to UNIX semantics and fails to locate a particular path component in the UNIX file system, it translates all slashes to underlines (_) and treats the string as a single name (e.g., "`Font/DA Mover`" becomes "`Font_DA Mover`").

If you must generate a full or relative pathname, it is best to use a colon as a pathname delimiter because this reduces the chance of ambiguity.

End-Of-Line Treatment

UNIX and Macintosh use different end-of-line characters (newline and carriage return, respectively). To allow both UNIX and Macintosh tools to manipulate files of type "TEXT," the A/UX File Manager transparently translates these characters when dealing with files residing on the / volume. Carriage returns are translated into newlines upon `_PBWrite` and newlines are translated into carriage returns upon `_PBRead`. Hence, Toolbox applications always see "`TEXT`" data in Macintosh format and UNIX applications always see "`TEXT`" data in UNIX format. To facilitate this transparency, it is important that applications set the file type immediately after a call to `_Create` (using either `_SetFileInfo` or `_SetCatInfo`).

File Permissions

Unlike AppleShare, UNIX file systems associate access permissions with files as well as directories. Directory permissions are mapped onto corresponding AppleShare file permission values and returned by `_GetDirAccess`. For unreadable files, the File Manager returns a special type and creator pair, namely creator "A/UX" and type "FILE" for data files, and type "EXEC" for executable files.

HFS-Specific File Information for UNIX Volumes

UNIX file systems lack certain HFS file attributes such as directory IDs and Finder information, which means files created outside of the Toolbox environment do not intrinsically have such attributes. When necessary, such information is provided by the File Manager through the use of a database in the blessed folder. When building its database, the File Manager synthesizes type and creator information for these UNIX files from the file's contents.

Directory IDs may, under some circumstances, change. For instance, if a particular UNIX volume is sometimes mounted in different areas of the / volume, the IDs for its directories change according to where it is mounted.

The File Manager tries to keep all files of type "`TEXT`" in "pure" UNIX format; consequently, if an application creates a file of type "`TEXT`" that has no resource information, then the type and creator association is maintained strictly through the database.

Miscellaneous UNIX Volume Details

The following caveats apply only to file accesses to the / volume:

- `_Allocate` and `_AllocContig` do nothing; applications should use `_SetEOF` instead.
- Do not directly access VCBs and FCBs. The UNIX external file system does not use these structures internally. Applications can use the File Manager calls instead (e.g. `_GetFCBInfo`).
- UNIX activity outside of the Toolbox environment may modify the / volume. The A/UX File Manager regularly changes the reported modification time of the / volume (as returned by `_GetVolInfo`) to stimulate applications to check if a directory they are displaying needs to be updated. Be sure to check for change the modification time of the particular directory before updating an application's display.
- Almost all File Manager traps intended for / may move or purge memory. The following list contains traps that could, now or in the future involve the Memory Manager to move memory (note that `_Read` and `_Write` do not cause any Memory Manager activity):

```
_MountVol
_OpenWD
_SetDir
_DirCreate
_GetCatInfo
_SetCatInfo
_CatMove
_GetDirAccess
_GetFileInfo
_SetFileInfo
_Open
_OpenRF
_Create
_Delete
_ReName
_SetVol     (if HFS-bit is set, as this implicitly involves a _SetDir )
_SetFilLock
_RstFilLock
_InitFS
```

- *Inside Macintosh*, Volume V-380, The File Manager, states that an `_Open` (and `_OpenRF`) call using `fsWrPerm` and `fsRdWrPerm` is retried as `fsRdPerm` in the case of read-only folders. Under A/UX 2.0, such calls return `fLckdErr` if the user does not have write permission for the particular file. Also, if that file resides on a UNIX file system which has been mounted as read-only, a `vLckdErr` is returned instead of `fLckdErr`. Both these problems are corrected in A/UX 2.0.1.

Virtual Memory

The A/UX 2.0 Toolbox is a virtual memory environment and is currently limited to 16 MB. The size of the environment defaults to the size of physical memory, but can be set at login time using the `TBMEMORY` environment variable. For example, a value for `TBMEMORY` of 10m results in a 10MB toolbox virtual memory environment.

24-Bit Environment

A/UX 2.0 provides a 24-bit, single application environment which may allow some applications which are not 32-bit clean to run under A/UX. Applications are still subject to limitations regarding direct hardware access and use of privileged instructions, but this environment may allow users to continue using older versions of software. Developers must not depend upon this environment for future development. In this 24-bit environment, virtual memory is limited to 8 MB.

Slot Manager

When A/UX 2.0 boots it checks each NuBus card. If the card contains a `PrimaryInit` (and a `SecondaryInit`) record, A/UX tries to execute the code. If this code does not follow the general A/UX guidelines (not 32-bit clean, it calls traps not supported...) then the kernel gets a panic message and the whole system hangs.

If the PrimaryInit and the SecondaryInit are not present, A/UX assumes that the card initializes itself or does not require initialization. Test for A/UX using `Gestalt` if the `PrimaryInit` contains code that will not work under A/UX.

Video Driver

A/UX 2.0 only supports well-behaved video driver control codes. The supported calls are documented in `/usr/include/mac/video.h`. The driver does a sanity check of the passed control calls. If the call is not supported or accepted, the control call is never sent to the video card.

The only Apple MacOS video driver control code not supported by the A/UX video driver is the `GetGamma, csCode 4`. `SetGamma, csCode 8`, is supported from A/UX 2.0.1 forward.

Developers interested to apply for control calls that are accepted under A/UX should contact MACDTS. The request should provide information about what csCode number the developer would like to have assigned, what functionality the control code contains, and what possible side effects this video control code has on the system.

Sound Manager

MacOS 6.0.7 supports the new `_SoundDispatch` trap starting from 6.0.7 forward. This trap is not implemented under A/UX 2.0.1 - even if the basic System release is 6.0.7. This means that programs which only test for System release level for `_SoundDispatch` availability will break with A/UX 2.0.1. Always test for the availability of the `_SoundDispatch` trap itself.

SCSI Manager

A/UX 2.0(.1) does not support the SCSI Manager. Thus the `_SCSIDispatch` trap is not implemented.

Script Manager

Starting with A/UX 2.0.1 all the Script Manager functionality and traps are supported.

Vertical Retrace Manager

`_AttachVBL and _DoVBLTask` traps are not implemented. These traps have to do with managing independent VBL queues for each video device. The A/UX kernel isolates the application from these devices - it handles the interrupts directly, and provides virtual vertical retrace interrupts via Unix signals.

`_AttachVBL` and `_DoVBLTask` are mostly used by system level software and interrupt handlers. Also the low memory global `jDoVBLTask` is not initialized.

A/UX Startup process and MacOS Systems

A/UX is booted from a special A/UX Startup application from MacOS. Some may have problems to actually boot A/UX from the Macintosh environment. The problem has to do with the A/UX Startup program and the Startup tools. These tools (launch,fsck, etc) consist of position dependent code because they are compiled with the A/UX C compiler and moved to the A/UX Startup environment.

Because the base address of these tools are `0x9FC00` (about 639k) it means that any INITs or other patches which increase the system heap above this limit will break the A/UX Startup shell and the tools. It can't either be changed out in the field. The only realistic workaround for the moment is to use the A/UX installed MacOS System when booting the computer for A/UX use later. This problem is subject to change in future.

Further Reference:

- *A/UX Toolbox: Macintosh ROM Interface*
- *Inside Macintosh*, Volume V, Compatibility Guidelines
- *Inside Macintosh*, Volume V, The File Manager
- Technical Note #117, Compatibility: Why & How
- *develop*, January 1990, "Compatibility: Rules for the Road"
- *UNIX Review*, June 1990, "UNIX as a Platform for Macintosh Applications"
- A/UX Device Driver Kit (APDA)
- *Designing Cards and Drivers for Macintosh II and Macintosh SE* , Second Edition

NFS is a trademark of Sun Microsystems, Inc.
UNIX is a registered trademark of AT&T

Macintosh
Technical Notes

Developer Technical Support

#230: Pertinent Information About the Macintosh SE/30

Revised by: Chris Knepper June 1989
Written by: Chris Knepper April 1989

This Technical Note discusses the Macintosh SE/30, items of interest to developers, and sources for further information.
Changes since April 1989: Corrected an error in the addresses of the video display buffers.

The Macintosh SE/30 is a modification of the original Macintosh SE concept. The SE/30 combines the modularity of the original SE with the capabilities of the larger Macintosh IIx. Although the name implies that the SE/30 borrows many characteristics from the SE, there are actually substantial differences between the two machines, and this Note addresses some of those differences.

Similarities Between the Macintosh SE and SE/30

The main similarities between the SE and the SE/30 are as follows:

- compact design
- power supply
- analog board
- rear housing
- SCSI support
- ADB support
- nine inch video display

Differences Between the Macintosh SE and SE/30

There are, however, many differences between the two machines. This section covers those differences with respect to their impact on developers.

CPU

The Motorola 68030 on the Macintosh SE/30 is clocked at 15.6672 MHz and provides both 32-bit data and address buses, both 256-byte instruction and data caches, and a built-in Paged Memory Management Unit (PMMU). The 68000 in the Macintosh SE is clocked at 7.83 MHz.

Although the 68030 is capable of a burst mode to more efficiently access contiguous blocks of memory, this feature is not enabled on the Macintosh SE/30. Enabling this feature would require significantly more complex control logic and faster (read "more expensive") RAM.

Coprocessor

The Motorola 68882 on the Macintosh SE/30 offers a full implementation of the IEEE Standard for Binary Floating-Point Arithmetic. The 68882 has an optimized MPU interface that provides up to 1.5 times the performance of the 68881. The Macintosh SE does not ship with a coprocessor (although third-party coprocessors are available).

ROM

The Macintosh SE/30 ROM is identical to that of the Macintosh IIx; it includes Color QuickDraw, the Slot Manager, and other features of the IIx ROM. It is composed of four 512 Kbit ROMs, for a total of 256K, and it is mounted on one ROM SIMM (ROM SIMMs are expandable to 2 MB).

System Software 6.0.3 (and later) patches to the ROM affect the Start Manager, the OS utilities, and the Sony driver.

RAM

The SE/30 includes eight RAM SIMM slots like the Macintosh II family and supports from 1 MB up to 128 MB (using 16 MB SIMMs), however, the current System Software only supports the first eight megabytes of RAM. The SE/30 also supports 4 MB DRAMs. For more information on memory configuration, see Macintosh Technical Note #176, Macintosh Memory Configurations.

Video

The Macintosh SE/30 video architecture is compatible with the SE: one-bit per pixel monochrome display with 342 lines of 512 pixels each. There is 64K of high-speed video display RAM to maximize video performance. The video provides dual display buffers of 21,888 bytes for fast page switching: the primary buffer starts at $FEE08040 and ends at $FEE0D5C0, and the alternate buffer starts at $FEE00040 and ends at $FEE055C0. The physical address of the video buffers simulates the NuBus address of slot $E on the Macintosh II family.

Developers need to be cautious with this video implementation, since a call to `_SysEnvirons` returns true for `hasColorQD` (since Color QuickDraw is implemented in the SE/30 ROM), but the default configuration only includes a single monochrome display.

As with any machine which supports Color QuickDraw, your application should test for the specific functionality which it needs, keeping in mind that different capabilities may be present on devices other than the main display. For example, an application which requires eight or more bits of color may do the following:

```
gotOne := FALSE;                                    {Assume we'll fail}
aDevice := GetMainDevice;                           {Get the first device}
WHILE (aDevice <> NIL) AND (NOT gotOne) DO
       IF (aDevice^^.gdPMap^^.pixelSize >= 8) AND {Do we have >= 8 bits?}
          (BitTest(@aDevice^^.gdFlags,15)) THEN    {And are we color?}
              gotOne := TRUE                        {Yes! We're done}
       ELSE
              aDevice := GetNextDevice(aDevice);    {Try next device}

IF gotOne THEN
              {We have a screen to use. Maybe put our window up
               within aDevice^^.gdRect}
```

Processor Direct Slot Expansion

The Processor Direct Slot (PDS) in the SE/30 is significantly different from the PDS in the SE. The 68000 PDS in the SE provides a 16-bit bus, whereas the 68030 PDS in the SE/30 provides both a 32-bit data bus and a 32-bit address bus. Access to the full 32-bit data bus allows for higher performance expansion cards than the 16-bit bus of the PDS in the SE. In addition, many of the expansion cards built for the PDS in the SE were hard-wired to the 7.83 MHz clock speed. Since the clock speed in the SE/30 is 15.6672 MHz, there are fundamental incompatibilities in clock speed, and therefore expansion card design.

The PDS in the SE/30 is a 120-pin, 32-bit PDS which provides both "common" and "machine-specific" signals. The common signals will be available across all 68030 PDS implementations, while the machine-specific signals will be available on future 68030 PDS implementations and may have new features added. On the SE/30, the machine-specific signals emulate equivalent signals on the NuBus expansion interface. This emulation means that expansion cards on the SE/30 may take advantage of the Slot Manager in ROM to communicate with the bus via a Declaration ROM on the card.

Connectors for the PDS may be obtained from AMP (part number 535022-1).

Prototyping cards for the PDS may be obtained from:

Creative Solutions
4701 Randolph, Suite 12
Rockville, Md 20852
Attn: Chris Colburn
(301) 984-0262

Disclaimer: This listing for Creative Solutions neither implies nor constitutes an endorsement by Apple Computer, Inc. If your company supplies these cards and you would like to be listed, contact us at the address in Technical Note #0.

The chassis design of the SE/30 simplifies expansion card installation as cards may be mounted vertically instead of horizontally, as in the SE. Because of this orientation, expansion cards can be installed without removing the logic board. In addition, there is more room for expansion cards in the SE/30 than in the SE.

System Software Requirements

The SE/30 requires System Software version 6.0.3 or later. Beginning with version 6.0.3, the installer shipped with the System Software includes a specific script for the SE/30.

Internal Floppy Drive

All configurations of the Macintosh SE/30 ship with an internal FDHD (Floppy Drive, High Density) floppy drive (a.k.a., SuperDrive) controlled by the SWIM controller chip. The SWIM chip is capable of supporting 720K and 1.44 MB Modified Frequency Modulation (MFM) formats (i.e., MS-DOS 3.5" disks), as well as the 400K and 800K Group Coded Recording (GCR) formats (Macintosh and Apple II 3.5" ProDOS disks) and the 1.4 MB MFM format (Macintosh 3.5" high-density disks). Note that special disks are required to take advantage of the 1.44 MB and 1.4 MB MFM formats. These disks have a square cutout in the top left corner to differentiate them from standard floppy disks. These disks may **not** be used in standard floppy drives (i.e., 400K and 800K) in the Macintosh family.

Although the SE is capable of supporting two internal 800K floppy drives, the SE/30 only supports a single internal FDHD drive.

External Floppy Drive

The SE/30 provides support for an external 800K floppy drive; it does not support the external 400K floppy drive or the external HD20 hard disk.

SCSI

The SE/30 uses the same 53C80 and interface logic as the IIx. This combination supports pseudo-DMA burst transfers, and SCSI performance matches that of the IIx.

_SysEnvirons

On the SE/30, `_SysEnvirons` version 2 returns 7 in the `machineType` field and 4 in the `processor` field. For more information about `_SysEnvirons`, see Macintosh Technical Note #129, _SysEnvirons: System 6.0 and Beyond.

Sound

The SE/30 uses the Apple Sound Chip, rather than the discrete sound circuitry of the SE. Although the circuitry provides stereo output to the speaker jack, as of System Software 6.0.3, stereo sound is **not** implemented, so true stereo is not yet available. The internal speaker of the SE/30 uses a mixed signal from both channels, whereas a Macintosh II uses a signal from only one channel.

Clock

The battery is not soldered to the motherboard and is replaceable.

General

All positive 5.0 volt outputs from the SE/30 (ADB, floppy drive, SCSI) are fuse protected from overloads. The maximum current that an external device connected to these ports can draw is 800 mA.

Upgrade Kits

Apple will offer several SE-to-SE/30 upgrade kits for current SE owners in the Spring of 1989. However, installation of these upgrade kits will prevent the owner from using any current SE expansion cards.

The first upgrade kit consists of the following:

- Logic board with 1 MB RAM
- Chassis
- EMI shroud
- Ferrite bead for power cable
- SE disk drive slot cover and retainer clip (for second floppy drive, if necessary)
- Owner's Manual

The dealer performing the upgrade is required to return the SE logic board with 1 MB RAM to Apple. A separate upgrade kit is available to upgrade the internal floppy drive to the FDHD. This kit is optional, since the SWIM chip on the SE/30 logic board is capable of controlling the 800K

floppy drive of the SE. However, upgraded SEs with two internal floppy drives will **not** be able to access the second drive, since the SE/30 only supports **one** internal floppy drive.

Software Compatibility

Apple's Product Quality and Support (PQ&S) department has tested over 100 of the most popular software packages on the SE/30 and has found that with the latest versions of these applications, 95% are completely compatible with the SE/30. Apple notified those developers whose tested packages were determined to be incompatible, and they are expected to announce upgrades in the near future.

Information on which packages were and were not compatible is **not** available, however, most of the incompatibilities were determined to be due to an application making assumptions about the hardware which were not true on the SE/30. In general, applications which call `_SysEnvirons` to determine the current hardware configuration are compatible.

For Further Information...

You can get further information about developing for the SE/30 from APDA. They offer the Macintosh SE/30 Developer Notes (part number M0061LL/A) for developers interested in producing hardware expansion cards, as well as a more general reference, *Designing Cards and Drivers for the Macintosh II and Macintosh SE*, (part number M7075).

Further Reference:

- *Inside Macintosh*, Volume V-1, Compatibility Guidelines
- Technical Note #129, _SysEnvirons: System 6.0 and Beyond
- Technical Note #176, Macintosh Memory Configurations.

Macintosh
Technical Notes

Developer Technical Support

#231: Macintosh Allegro Common Lisp Features

Revised by: Paul Snively
Written by: Guillermo Ortiz

February 1990
April 1989

This Technical Note describes some known problems and provides solutions to these problems for the Macintosh Allegro Common Lisp™ package which is available from Apple Computer, Inc. You should note, however, that although Apple acquired Coral Software and is selling Macintosh Allegro Common Lisp, Apple is not currently distributing any other products which had been developed or previously sold by Coral Software.
Changes since April 1989: Noted 1.3.1 documentation errors, corrected erroneous floating-point patch for version 1.2.2, updated 1.2.2 information which is not relevant to 1.3.1, corrected APDA part number, added examples of high-level printing functions in 1.3.1, an array-dialog-item example, and information explaining how to get the Victoria-Day release of Portable Common LOOPS (PCL) to compile correctly under 1.3.1.

The current supported version of the Macintosh Allegro Common Lisp package (MACL) is 1.3.1; if you have an earlier version of this product, you should obtain an upgrade through APDA (part number M0229LL/C), as Apple only supports the current version.

Some Known Problems and Solutions

- **Typographical errors in the 1.3.1 documentation.**
  The following symbols, documented in the "Menus" chapter, are missing a hyphen (-) in the documentation. Their proper names are as follows:

```
add-menu-items              menu-item-enable
remove-menu-items           menu-item-enabled-p
menu-items                  menu-item-check-mark
*menu-item*                 set-menu-item-check-mark
menu-item-title             menu-item-style
set-menu-item-title         set-menu-item-style
menu-item-disable           *window-menu-item*
```

  The following keywords are misspelled in the "Menus" chapter and do not appear in the index:

```
:menu-items                 :menu-item-colors
:default-menu-item-title    :menu-item-checked
:menu-item-title
```

  The following symbol is misspelled only in the index:

```
menu-update
```

  The following symbol is misspelled only in its descriptive paragraph:

```
menu-item-update
```

- **Old versions crash on Macintoshes with a 68030 processor.**
  Versions of MACL prior to 1.2.2 may crash when running on 68030 machines because they are not compatible with this processor. You must upgrade to version 1.2.2 or later to solve this problem.

- **Some 1.2.2 packages do not run in the background.**
  Some MACL 1.2.2 packages which were distributed by APDA did not have the `'canBackground'` flag in the `'SIZE'` resource set, and will prevent background operation when running under MultiFinder. You can check for this problem by launching ResEdit and opening the `'SIZE' ID=-1` resource in MACL. If the `'canBackground'` flag is not set, you should set it.

- **Practice safe hex.**
  Due to its segment numbering scheme, MACL is very sensitive to viral infections. If things stop working for no apparent reason, check for viruses. You did back it up, didn't you?

- **Shut Down and Restart do not work in 1.2.2**
  Under MultiFinder, if you select Shut Down or Restart from the Special menu, the "going away" process stops with MACL 1.2.2 until you Quit it manually. This feature is fixed in 1.3.1.

- **Color dialogs and menus are not supported in 1.2.2.**
  MACL 1.2.2 does not support color dialogs, alerts, or menus. This feature is present in 1.3.1.

- **Only RAM pointers please.**
  Current versions of MACL cannot handle pointers into ROM or NuBus memory.

- **More memory?**
  Current versions of MACL are limited to supporting eight megabytes of memory. Future versions will support as much memory as the Macintosh OS supports.

- **Problems displaying PICTs on windows with 1.2.2.**
  There is a problem with clipping when displaying pictures that require resizing in 1.2.2. This problem has been fixed in 1.3.1, which utilizes a completely new view system modeled after MacApp. For developers still working with 1.2.2, the way to work around this problem is to replace the definition of `START-PICTURE` in the file QuickDraw.lisp with the following:

```
(defobfun (start-picture *window*) (&optional left top right bottom)
  (if (rref wptr window.picsave)
   (error "A picture may not be started for window: ~a.
           since one is already started" (self)))
 (unless left (setq left (rref wptr window.portrect)))
 (with-rectangle-arg (r left top right bottom)
  (with-port wptr
    (_cliprect :ptr r)
    (have 'my-hPic (_OpenPicture :ptr r :ptr))))
 nil)

;;<this just adds a (_cliprect :ptr r) to the old definition>
```

- **1.2.2 crashes on Macintoshes with a 68882 coprocessor.**
  Some old versions of MACL can crash on Macintoshes with a 68882 floating point coprocessor. Certain errors, such as a floating point divide-by-zero, are not caught and crash the machine instead of being reported as Lisp errors. This bug is fixed in MACL 1.3.1, but developers still using versions prior to 1.3.1 can include the following patch in the file init.Lisp, so it gets executed before anything else:

```
(in-package "CCL")

(defun validate-fp-handler (handler)
  "If the HANDLER argument appears valid, return it.  Otherwise,
   make a new one which Does The Right Thing.
   Note that the kernel will restore system floating-point
   exception handlers on exit, so we don't worry about that here."
  (let* ((old-addr (%ptr-to-int handler))
         (words (list
                 #o026417 ; move.l sp,-(a6)
                 #o171447 ; fsave -(sp)
                 #o027400 ; move.l d0,-(sp)
                 #o027401 ; move.l d1,-(sp)
                 #o070000 ; moveq #0,d0
                 #o010057 ; move.b 9(sp),d0
                 #o000011
                 #o004367 ; bset #27,8(sp,d0.w)
                 #o000033
                 #o000010
                 #o171000 ; fmove.l fpsr,d0
                 #o124000
                 #o031074 ; move.w #$3400,d1
                 #o032000
                 #o141100 ; and.w d0,d1
                 #o063012 ; bne.s @1
                 #o021037 ; move.l (sp)+,d1
                 #o020037 ; move.l (sp)+,d0
                 #o171537 ; frestore (sp)+
                 #o054116 ; addq #4,a6
                 #o047163 ; rte
                 #o027136 ; @1: move.l (a6)+,sp
                 #o041247 ; clr.l -(sp)
                 #o171537 ; frestore (sp)+
                 #o171000 ; fmove.l d0,fpsr
                 #o104000
                 #o047371 ; jmp <old-handler>
                 (logand (ash old-addr -16) #xff)
                 (logand old-addr #xffff)))
         (len (length words)))
    (unless (= (%get-signed-word handler)
               (car words))
      (setq handler (%register-trap #xa11e 384 (* 2 len)))  ;(_Newptr:d0 (* 2 len):a0)
      (dotimes (i len)
        (%put-word handler (pop words) (+ i i))))
    handler))

(defun ccl-using-fpu-p ()
   (not (= 0 (%get-byte (%get-ptr (%int-to-ptr #x904)) #x-130))))

(when (ccl-using-fpu-p)
  (let* ((addr (%int-to-ptr #xc8)))
    (%put-ptr addr (validate-fp-handler (%get-ptr addr))))
  t)
```

- **High-level printing functions.**
  The high-level printing functions that were available in 1.2.2 are no longer available in 1.3.1. Following is the code necessary to implement hardcopy for `*window*` objects:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;hardcopy.lisp
;;
;;
;;Copyright 1988-99 Apple Computer, Inc.  All Rights Reserved.
;;
;; defines a very basic printing routine for windows
;;
;; This code sets the window's wptr to a printer grafport
;; and then calls view-draw-contents
;;
;; This code does allow printing of Dialogs
;;


(eval-when (eval compile)
  (require 'traps)
  (require 'records)
  (defconstant $PrintErr #x944)
  (defconstant $prJob.bJDocLoop (+ 62 6))
  (defconstant $iPrStatSize 26)
  (defconstant $bSpoolLoop 1)
  (defconstant $err-printer 94)
  (defconstant $err-printer-load 95)
  (defconstant $err-printer-start 97)
)

(defun prchk (&optional (errnum $err-printer)
                        &aux (print-error (%get-signed-word $PrintErr)))
   (unless (zerop print-error)
      (ccl::%signal-error errnum print-error)))

(defobfun (set-view-wptr *view*) (new-wptr)
  (setf (objvar wptr) new-wptr)
  (let ((subviews (objvar view-subviews)))
    (dotimes (index (length subviews))
      (ask (aref subviews index)
        (set-view-wptr new-wptr)))))

(defobfun (window-hardcopy *window*) ()
  (window-select)
  (unwind-protect
    (with-cursor *arrow-cursor*
      (_PrOpen)
      (prchk $err-printer-load)
      (let ((pRec (get-print-record)))
        (when (_PrJobDialog :ptr pRec :boolean)
          (let ((*hc-page-open-p* nil) (ccl::*inhibit-error* t) err)
            ;_PrOpenDoc puts up a dialog window which causes the event system
            ;to get confused.  So we do the whole thing without interrupts, and
            ;make sure to clean up before handling errors.
            (declare (special *hc-page-open-p* ccl::*inhibit-error*))
            (setq err (catch-error-quietly
                       (without-interrupts
                        (with-port (_PrOpenDoc :ptr pRec :long 0 :long 0 :ptr)
                          (let ((window-ptr wptr)
                                (hardcopy-ptr (ccl::%getport)))
```

```
                                (unwind-protect
                                  (with-dereferenced-handles ((ppRec pRec))
                                    pprec
                                    (prchk $err-printer-start)
                                    (unwind-protect
                                      (progn
                                        (set-view-wptr hardcopy-ptr)
                                        (_PrOpenPage :ptr hardcopy-ptr :long 0)
                                        (view-draw-contents)
                                        (_PrClosePage :ptr hardcopy-ptr))
                                      (set-view-wptr window-ptr)))
                                  (_PrCloseDoc :ptr hardcopy-ptr))))
                              (when (eq (%hget-byte pRec $prJob.bJDocLoop)
                                        $bSpoolLoop)
                                (prchk)
                                (%stack-block ((StRec $iPrStatSize))
                                 (_PrPicFile :ptr pRec :long 0 :long 0 :long 0 :ptr StRec))
                                (prchk)))))
                t))))
        (_PrClose)))

;;unfortunately, this doesn't work for dialogs
(defobfun (window-hardcopy *dialog*) ()
  (message-dialog "Cannot print of dialogs at this time"))

#|
(require 'quickdraw)
(setq foo (oneof *window*))

(defobfun (view-draw-contents foo) ()
  (frame-rect 10 10 100 100)
  (usual-view-draw-contents))

(setq bar (oneof *view*
                 :view-container foo
                 :view-position #@(150 150)))

(defobfun (view-draw-contents bar) ()
  (paint-oval 10 10 100 100)
  (usual-view-draw-contents))

(ask foo (window-hardcopy))
|#
```

- **I want to use a grapher with MACL.**
  A simple grapher is included as an example source file with 1.3.1.

- **Missing array-dialog-item example in 1.3.1 Examples folder.**
  The 1.3.1 documentation mentions an array-dialog-item example, but it is missing from the Examples folder on the disk. Following is the missing example:

```
;;;;;;;;;;;;;;;;;;;;
;;
;;  array-dialog-items
;;
;;  ©1989, Apple Computer, Inc
;;
;;
;;  a subclass of table-dialog-items used to display two-dimensional arrays
;;
```

```
(in-package :ccl)

(export '(*array-dialog-item* table-array set-table-array) 'ccl)

(defobject *array-dialog-item* *table-dialog-item*)

(defobfun (exist *array-dialog-item*) (init-list)
  (let* ((the-array (getf init-list :table-array (make-array '(0 0))))
         (dims (array-dimensions the-array)))
    (unless (eq (length dims) 2)
      (error "table-array ~s is not of rank ~d" the-array 2.))
    (have 'my-array the-array)
    (usual-exist (init-list-default
                  init-list
                  :table-dimensions (make-point (car dims) (cadr dims))))))

(defobfun (cell-contents *array-dialog-item*) (h &optional v)
  (unless v
    (setq v (point-v h)
          h (point-h h)))
  (aref (objvar my-array) h v))

(defobfun (table-array *array-dialog-item*) ()
  (objvar my-array))

(defobfun (set-table-array *array-dialog-item*) (new-array)
  (setf (objvar my-array) new-array)
  (inval-dialog-item))


#|
(setq table (oneof *array-dialog-item*
                   :table-array #2a((a1 b1 c1)
                                    (a2 b2 c2)
                                    (a3 b3 c3))))
(oneof *dialog*
       :dialog-items (list table))

(ask table (set-table-array #2a((x1 y1 z1)
                                (x2 y2 z2)
                                (x3 y3 z3))))

|#
```

- **Common Lisp Object System (CLOS).**
  Future versions of MACL will support an Apple implementation of CLOS, but you can use PCL, a portable implementation of CLOS, until that time. PCL is available from various sources, including APDA.

  If you have the Victoria-Day release of PCL, the following changes to the source code allow it to compile successfully under MACL 1.3.1:

  In file defsys.lisp:

  Find the `defvar` for `*pcl-directory*`. Within it, find the conditional for MACL (`#+:coral`). Change the pathname parameter to point to your PCL folder (e.g., `"ccl;PCL:"`). Also find the `let` of `files-renamed` and change its binding to `nil`.

In file coral-low.lisp:

Comment out both the `ccl::add-transform` and its `inline` proclamation. Neither is helpful in 1.3.1 (in fact, the `add-transform` is broken with respect to 1.3.1).

In file fin.lisp:

Immediately before the closing "`); End of #+:coral`" that you find near the end of the file, add:

```
(defun print-uvector-object (obj stream &optional print-level)
  (declare (ignore print-level))
  (print-object obj stream))

(pushnew (cons 'ccl::funcallable-instance #'print-uvector-object)
         ccl:*write-uvector-alist* :test #'equal)
```

In addition to these code changes, there are some environmental settings that are useful or necessary when compiling Victoria-Day PCL. You should use the following settings:

```
(setq *WARN-IF-REDEFINE-KERNEL* nil)
(setq *COMPILER-WARNINGS* nil)
(setq *FASL-COMPILER-WARNINGS* nil)
(setq *FAST-EVAL* nil)
```

These settings eliminate loads of warnings that you would otherwise get when compiling or loading PCL. In particular, you must bind or assign `nil` to `*FAST-EVAL*` for the file test.lisp to load correctly.

Allegro Common Lisp is a trademark of Franz Inc.



#232: Strip With _OpenResFile and _OpenRFPerm

Written by: Byron Han April 1989

This Technical Note discusses a bug in `_OpenResFile` and `_OpenRFPerm` which can cause system crashes and what you can do to avoid this problem.

The traps `_OpenResFile` and `_OpenRFPerm` call some common code in 128K and later ROMs which was affected by some system patches for early print drivers. The problem is that the common code checks an attribute bit in the pointer to the string name to see if it is a dereferenced handle. If the pointer has the resource attribute bit set, the Resource Manager assumes that it is a dereferenced handle and calls `_RecoverHandle`. This usually works, but when the string is embedded in a code resource, the Resource Manager calls `_RecoverHandle` with an invalid master pointer.

Note: In MPW C, this bug is **not** a problem, unless you use either the `-b`, `-b2`, or `-b3` options, which embed string constants in the code segment. If you use these options, you must deal with this bug.

The following code fragments give an example of this bug:

MPW Pascal

```
VAR
    fileName : Str255;
    ref : INTEGER;

BEGIN
    fileName := 'This File';
    ref := OpenResFile(fileName);
END
```

MPW C

```
Str255          fileName;
short int       ref;

fileName = 'This File';
ref = OpenResFile(fileName);
```

Calling _StripAddress on the pointer to the filename prior to calling _OpenResFile or _OpenRFPerm solves the problem:

MPW Pascal

```
VAR
    fileName : Str255;
    ref : INTEGER;

BEGIN
    fileName := 'This File';
    ref := OpenResFile(StringPtr(StripAddress(@fileName))^);
END;
```

MPW C

```
Str255        fileName;
short int     ref;

fileName = 'This File';
ref = OpenResFile((StringPtr)StripAddress((Ptr)fileName));
```

By always calling _StripAddress before calling _OpenResFile or _OpenRFPerm, you will not have to deal with this problem, life will be good, and you will be able to rest a bit easier.

Further Reference:

- Technical Note #212, The Joy of Being 32-Bit Clean
- Technical Note #213, _StripAddress: The Untold Story

Macintosh
Technical Notes

Developer Technical Support

#233: MultiFinder and _SetGrowZone

Written by: Andrew Shebanow June 1989

MultiFinder patches the `_SetGrowZone` trap, and this patch can cause your program to crash if you attempt to save and restore the grow zone procedure.

MultiFinder gives each application its own heap in which to run. Because it wants to do some fairly tricky memory management, MultiFinder installs its own grow zone procedure (`gzProc`) in the application heap, and patches `_SetGrowZone` to store your application's `gzProc` in a temporary variable inside of itself.

A problem arises when you want to allocate some memory **without** invoking the application's `gzProc`. This can be useful if you are writing a library of routines that does its own internal caching, and you do not want that cache to purge the application's reserved memory. Let's say that to do this, you write a pair of routines, `KillGZProc` and `RestoreGZProc`, which look like this:

```
#include <Memory.h>

GrowZoneProcPtr savedGZProc;

pascal void KillGZProc(void)
{
        THZ myZone;

        myZone = GetZone();
        /*      since there is no GetGrowZone trap, we have to pull it directly
                from the zone header (Ugh! Very gross!) */
        savedGZProc = myZone->gzProc;
        /* we don't want a grow zone proc */
        SetGrowZone( (GrowZoneProcPtr) nil);
}

pascal void RestoreGZProc(void)
{
        /* set to saved value */
        SetGrowZone(savedGZProc);
}
```

Now let's say that you bracket your call to `_NewHandle` with these two routines. When MultiFinder is active, you get the following:

- When the application starts, you set your `gzProc` to the routine `MyGZProc`. MultiFinder stores the procedure pointer inside of your application's MultiFinder data area.
- You call `KillGZProc`. The global variable `savedGZProc` now contains a pointer to **MultiFinder's** `gzProc`, which MultiFinder installed in your zone header before your application started.

- You do your memory allocation, and your `gzProc` (`MyGZProc`) doesn't get called, just as you intended.
- You call `RestoreGZProc`, which stores a pointer to MultiFinder's `gzProc` in your application's MultiFinder data area.
- The next time you do a memory allocation that causes the `gzProc` to be called, MultiFinder's `gzProc` will be called. One of the things this `gzProc` does is to see if there is a valid `gzProc` stored in your application's MultiFinder data area. If there is a valid `gzProc`, it gets called. But the `gzProc` in your application's MultiFinder data area is MultiFinder's `gzProc`, so we go into an infinite loop. Oops...

The only solution to work around this problem is to avoid reading the value of the `gzProc` out of the zone header, since it isn't valid when MultiFinder is active. (Reading the fields of the zone header is dangerous, compatibility wise as well.) Your application should only have one grow zone procedure, so you should change your `KillGZProc` and `RestoreGZProc` to restore your application's grow zone procedure directly. The corrected code would look like the following:

```
#include <Memory.h>

pascal long MyGrowZone(Size cbNeeded);

pascal void KillGZProc(void)
{
        /* we don't want a grow zone proc */
        SetGrowZone( (GrowZoneProcPtr) nil);
}

pascal void RestoreGZProc(void)
{
        /* set to my routine */
        SetGrowZone(MyGrowZone);
}
```

As you can see, the code is simpler, though not quite as flexible, but at least it won't throw your machine for a loop.

Further Reference:

- *Inside Macintosh*, Volume II, Memory Manager
- *Programmers Guide To MultiFinder* (APDA)
- Technical Note #158, MultiFinder Questions
- Technical Note #205, MultiFinder Revisited
- Technical Note #212, The Joy Of Being 32-Bit Clean

Macintosh
Technical Notes

Developer Technical Support

#234: NuBus Physical Designs—Beware

Revised by: Rich Collyer December 1989
Written by: Rich Collyer June 1989

This Technical Note discusses the possible problems you might run into while designing a NuBus™ card. It covers some of the specifications which, if not followed, will have problems with current Macintosh machines, and possibly future machines.
Changes since June 1989: Added warnings about the no component area and full-size NuBus cards.

If you are making a NuBus card for the Macintosh II family of computers, then you have to be very careful to follow the physical specifications which are listed in the NuBus specifications (IEEE P1196). There are two areas where some developers have run into problems. The first problem has to do with not positioning the external connector properly. The result is that some products have problems with the external hole on the back of the Macintosh IIcx. The second problem has to do with developers who run ribbon cables over the top of their boards to connect two boards. If a slot is not cut into the top of the board to allow the ribbon cable to sit below the top of the card, then the boards will have problems in our machines.

External Connector

The NuBus specification allows for an external connector plastics opening of only 74.55 mm x 11.90 mm. The Macintosh II and IIx allowed a significantly larger hole than the specification (80.00 mm x 17.00 mm) and some developers incorrectly assumed that Apple would continue to allow for this larger size. When the Macintosh IIcx came out, these boards were incompatible, since the IIcx only allows for an external opening of 75.61 mm x 14.00 mm. This opening is still larger than the IEEE specification. We could shrink this size all the way to the limit of the NuBus specification in future machines. If you stay within the limits which are set down in the NuBus specification, then you should not have any problems with any of our machines.

There is one other important dimension which changed in the Macintosh IIcx; this is the intercard spacing. In the Macintosh II and IIx, the intercard spacing is set to the minimum space allowed by the NuBus specification (22.86 mm). In the Macintosh IIcx this dimension was expanded to 24.13 mm. Figure 1 shows the connector opening and intercard spacing for the Macintosh IIcx.

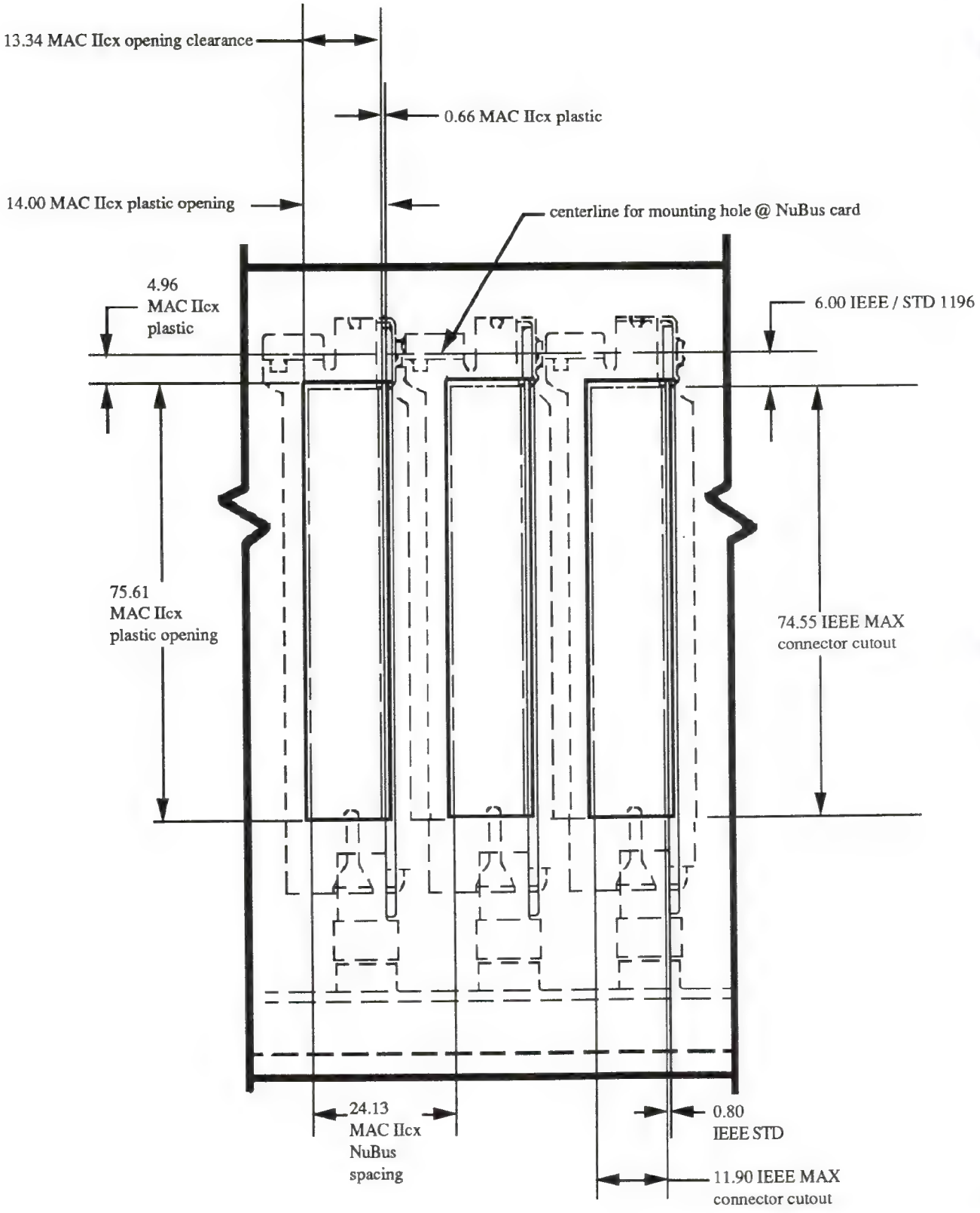


Figure 1–Macintosh IIcx External Connector Opening and Intercard Spacing

Internal Connector

Several NuBus card developers have the need to connect two boards. The NuBus specification allows for this need with an auxiliary connector at the top of the card and next to the no component area. To connect the cards, you need to use a ribbon cable. The cable is run over the top of the card as demonstrated in Figure 2. The problem occurs when the ribbon cable is run over the top of a card and is not given a slot into which to drop.

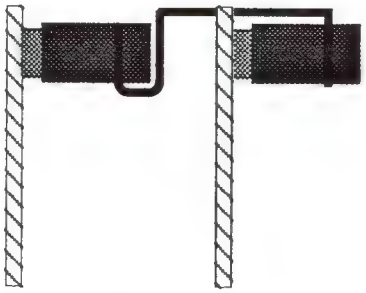


Figure 2–Side View of Internal Connection

Figure 3 is an example of the **wrong** way to make your internal connector. The ribbon cable will not fit over top of the NuBus card; you must make a slot at the top of your card for the ribbon cable. Refer to Figure 4 for an example of the correct way to make your internal connector.

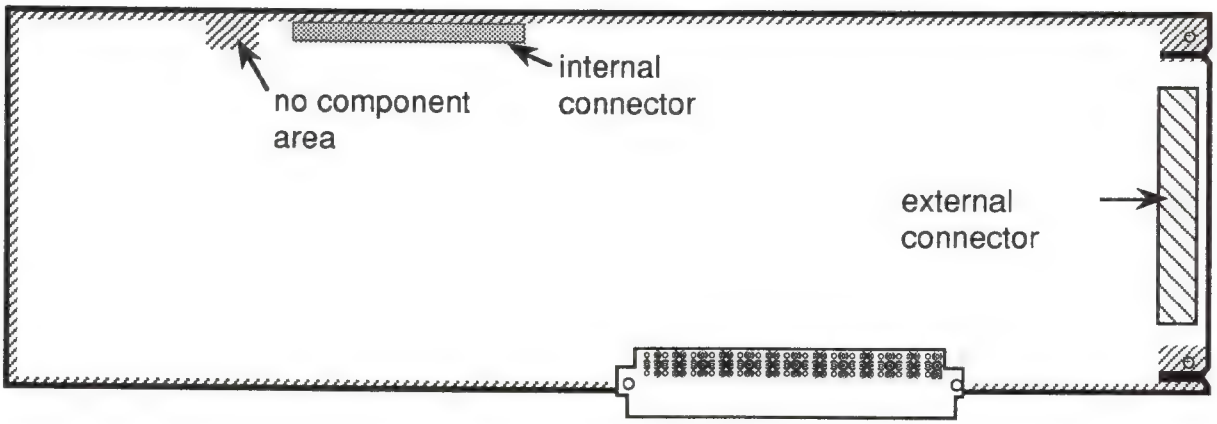


Figure 3–The Wrong Way

If you cut a slot at the top of your NuBus card, you will not have problems with future Macintosh computers which utilize the NuBus standard. The slot needs to be deep enough for the cable to be flush with the top of the card.

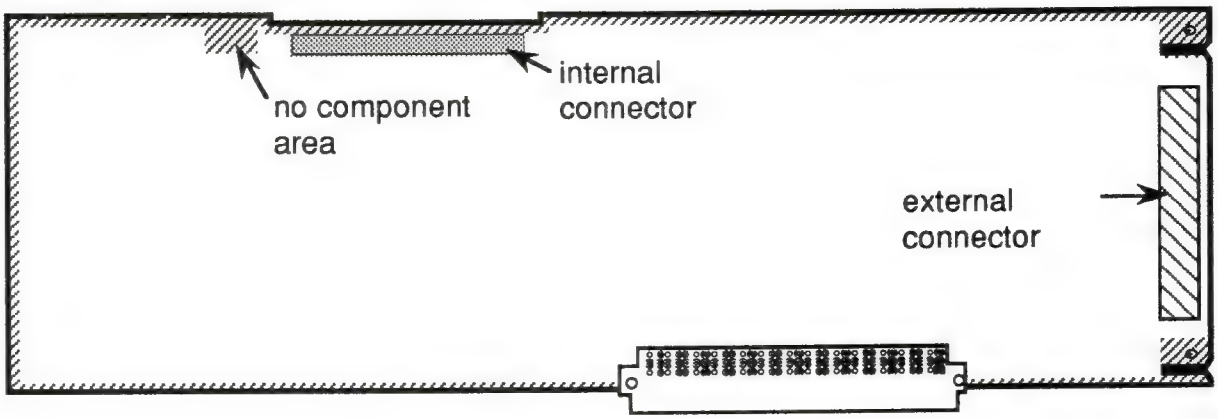


Figure 4–The Correct Way

The internal connector must not have any parts which extend into the "no component area." This means that if your connector has lock & eject tabs (like the internal SCSI connector) then the tabs must be below the "no component area."

The no component area is defined as the area of the card onto which you cannot put any parts. The lid of the Macintosh II family of computers has two fingers which hold the NuBus cards into place. These fingers are needed for stability, and they help to ensure that the cards will not be damaged in the event that the computer is knocked around. If there are components in the no component area, then the fingers will either break the components, or the lid will not sit correctly.

No Component Area

The no component area is not just an area in which you should not mount parts, but it is also a three-dimensional area. As such, the no component area covers the surface of the board as well as the distance to the next card (22.86 mm). This means that you must not violate this space with either mounted parts or daughter boards.

Full-Size NuBus Cards

It is important to test all full-size NuBus cards in the Macintosh modular platforms (e.g., IIcx and IIci). A full-size NuBus card extends 326.6 mm and might interfere with the NMI and reset buttons in these machines. Most cards should not have this problem, however, developers who find this problem with their cards should contact Developer Technical Support at the address listed in Technical Note #0.

Conclusion

The moral of this story is that you should always follow the NuBus specifications which deal with the physical dimensions of your cards, even if Apple allows for more space in certain models of our machines.

If your board violates any of the NuBus specifications, or if you have run a ribbon cable over the top of your card, then you need to seriously consider redesigning your board.

Further Reference:

- *NuBus—A Simple 32–Bit Backplane Bus P1196 Specification*

NuBus is a trademark of Texas Instruments

Macintosh
Technical Notes

Developer Technical Support

#235: Cooperating with the Coprocessor

Revised by: Jim Reekes
Written by: Jim Reekes

October 1990
June 1989

The use of the 68881 or 68882 coprocessor is usually handled by the SANE package or by a development system's libraries. Some developers may wish to use the coprocessor during special circumstances, such as at interrupt level or installing their own hardware floating point exception handlers. In these two situations, there are special requirements that must be met. These requirements will require floating–point assembly code and are discussed in this Technical Note.
Changes since June 1989: Noted that the new Sound Manager no longer uses floating-point numbers at interrupt time.

Witnessing the Problem

If you see the message "Spurious Interrupt of the Uninitialized Interrupt vector" in MacsBug or the message "Unassigned Interrupt #$00D (format 9)" from TMON™, you should suspect a floating point protocol violation. This can be caused by improper usage of floating point instructions at interrupt level or by attempting to handle hardware floating point exceptions incorrectly.

Interrupting the Coprocessor

If you attempt to use the coprocessor at interrupt level, you may be interrupting a floating point processor. You must save the coprocessor's state before executing any floating point instructions, and, of course, restore it later during the interrupt routine. This requires assembly code since there is no convenient way to do it in a high–level language.

There is a protocol that must be followed. The first floating point instruction must be an `FSAVE`. This instruction suspends the execution of any operation in progress and saves the internal state. The number of bytes required in this operation depends on the state it is in, and it can be up to 216 bytes. If any floating point registers are to be used, they also must be saved with the `FMOVE` instruction. After performing the interrupt routine, the `FRESTORE` instruction is used to restore the floating point state.

```
VBLProc     FSAVE       -(SP)           ; save the FP state
            FMOVEM.X    FPRegs,-(SP)    ; save the FP regs we use
            …
            …                           ; your interrupt code
            …
VBLExit     FMOVEM.X    (SP)+,FPRegs    ; replace the FP regs we used
            FRESTORE    (SP)+           ; restore the FP state
            RTS
```

Note that the coprocessor may not be in a condition to be interrupted, and the `FSAVE` instruction will halt the main processor until such a condition can be met. To give an idea on the time required

for saving the coprocessor's state, the FSAVE or FRESTORE instructions can take approximately 900 cycles to execute. This is about 50 times slower than a MOVE instruction. Considering the length of time it takes to perform this necessary protocol, it may not be desirable to use floating–point math at interrupt level. As an alternative, investigate the possibility of using the Toolbox routines for Fixed and Frac numbers.

Handling Floating Point Exceptions

It is possible, and sometimes desirable, for applications to install their own hardware floating–point exception handlers. The MPW '881 SANE libraries provide routines for applications to do so. If an application is going to use this mechanism to catch exceptions such as underflow, overflow, or divide by zero, then it must follow the minimal protocol as shown in the following example.

```
Handler     FSAVE       -(SP)           ; save the FP state
            MOVE.B      (SP),D0         ; first byte of the state frame
            BEQ         NULL            ; branch if NULL state
            CLR.L       D0              ; clear data register
            MOVE.B      1(SP),D0        ; load state frame size
            BSET        #3,(SP,D0)      ; set bit 27 or BIU
            …
            …                           ; your exception code
            …
Null        FRESTORE    (SP)+           ; restore the FP state
            RTE                         ; return from exception
```

Other Issues

Debugging floating–point routines with MacsBug, SADE, and TMON may cause a protocol violation. MacsBug 6.1, and earlier, do not perform the FSAVE and FRESTORE surrounding floating–point instructions at interrupt level. As of TMON 2.8.4, it has not handled floating-point instructions. Hardware floating–point exception handlers and interrupt routines using floating–point instructions require assembly coding.

You can witness a protocol violation in another situation. This is when using the Sound Manager in System Software 6.0.5 and earlier. This Sound Manager calls SANE at interrupt time. If an application is using the coprocessor **and** this Sound Manager is running, it is very likely to interrupt the coprocessor. This problem has been resolved in the new Sound Manager, which was originally released in System Software 6.0.6. The new Sound Manager no longer uses floating-point numbers at interrupt level; it replaces them with Fixed and Fract types.

Further Reference:

- *Apple Numerics Manual*, Second Edition
- *Motorola MC68881/MC68882 User's Manual*
- MPW reference manuals
- Technical Note #146, Notes on MPW's –mc68881 Option
- Technical Note #236, Speedy the Math Coprocessor

TMON is a trademark of ICOM Simulations, Inc.

Macintosh
Technical Notes

Developer Technical Support

#235: Cooperating with the Coprocessor

Written by: Jim Reekes June 1989

The use of the 68881 or 68882 coprocessor is usually handled by the SANE package or by a development system's libraries. Some developers may wish to use the coprocessor during special circumstances, such as at interrupt level or installing their own hardware floating point exception handlers. In these two situations, there are special requirements that must be met. These requirements will require floating–point assembly code and are discussed in this Technical Note.

Witnessing the Problem

If you see the message "Spurious Interrupt of the Uninitialized Interrupt vector" in MacsBug or the message "Unassigned Interrupt #$00D (format 9)" from TMON™, you should suspect a floating point protocol violation. This can be caused by improper usage of floating point instructions at interrupt level or by attempting to handle hardware floating point exceptions incorrectly.

Interrupting the Coprocessor

If you attempt to use the coprocessor at interrupt level, you may be interrupting a floating point processor. You must save the coprocessor's state before executing any floating point instructions, and, of course, restore it later during the interrupt routine. This requires assembly code since there is no convenient way to do it in a high–level language.

There is a protocol that must be followed. The first floating point instruction must be an FSAVE. This instruction suspends the execution of any operation in progress and saves the internal state. The number of bytes required in this operation depends on the state it is in, and it can be up to 216 bytes. If any floating point registers are to be used, they also must be saved with the FMOVE instruction. After performing the interrupt routine, the FRESTORE instruction is used to restore the floating point state.

```
VBLProc     FSAVE       -(SP)          ; save the FP state
            FMOVEM.X    FPRegs,-(SP)   ; save the FP regs we use
            …
            …                          ; your interrupt code
            …
VBLExit     FMOVEM.X    (SP)+,FPRegs   ; replace the FP regs we used
            FRESTORE    (SP)+          ; restore the FP state
            RTS
```

Note that the coprocessor may not be in a condition to be interrupted, and the FSAVE instruction will halt the main processor until such a condition can be met. To give an idea on the time required for saving the coprocessor's state, the FSAVE or FRESTORE instructions can take approximately 900 cycles to execute. This is about 50 times slower than a MOVE instruction. Considering the length of time it takes to perform this necessary protocol, it may not be desirable to use floating–

point math at interrupt level. As an alternative, investigate the possibility of using the Toolbox routines for `Fixed` and `Frac` numbers.

Handling Floating Point Exceptions

It is possible, and sometimes desirable, for applications to install their own hardware floating–point exception handlers. The MPW '881 SANE libraries provide routines for applications to do so. If an application is going to use this mechanism to catch exceptions such as underflow, overflow, or divide by zero, then it must follow the minimal protocol as shown in the following example.

```
Handler     FSAVE       -(SP)          ; save the FP state
            MOVE.B      (SP),D0        ; first byte of the state frame
            BEQ         NULL           ; branch if NULL state
            CLR.L       D0             ; clear data register
            MOVE.B      1(SP),D0       ; load state frame size
            BSET        #3,(SP,D0)     ; set bit 27 or BIU
            …
            …                          ; your exception code
            …
Null        FRESTORE    (SP)+          ; restore the FP state
            RTE                        ; return from exception
```

Other Issues

Debugging floating–point routines with MacsBug, SADE, and TMON may cause a protocol violation. MacsBug 6.1, and earlier, do not perform the `FSAVE` and `FRESTORE` surrounding floating–point instructions at interrupt level. TMON 2.8.1 does not handle floating–point instructions. Hardware floating–point exception handlers and interrupt routines using floating–point instructions will require assembly coding.

You will witness a protocol violation in another situation. This is when using the Sound Manager in System 6.0.3 and earlier. The Sound Manager calls SANE at interrupt time incorrectly. If an application is using the coprocessor **and** the Sound Manager is running, it is very likely that the coprocessor will be interrupted. This will be fixed in the next release of the Sound Manager.

Further Reference:

- *Apple Numerics Manual*, Second Edition
- *Motorola MC68881/MC68882 User's Manual*
- MPW reference manuals
- Technical Note #146, Notes on MPW Pascal's –mc68881 Option
- Technical Note #229, A/UX 1.1 Toolbox Bugs
- Technical Note #236, Speedy the Math Coprocessor

TMON is a trademark of ICOM Simulations, Inc.

Macintosh
Technical Notes

Developer Technical Support

#236: Speedy the Math Coprocessor

Written by: Rich Collyer June 1989

This Technical Note presents an overview of the 68881 and 68882 math coprocessors, and it covers general information about the chips as well as how using the chips directly can help speed your math–intensive code.

Introduction

Generally we don't recommend that you assume the existence of specific hardware. However, if your program does proper feature checking using `_SysEnvirons` and there is a Floating–Point Unit (FPU) available, than you can use code which will run your math intensive code much faster. This Technical Note is basically a condensed version of the *Motorola MC68881/MC68882 Floating–Point Coprocessor User's Manual.* I will cover some of the basics of what the chips can do, their differences, and how to take advantage of what they have to offer.

If `_SysEnvirons` returns `hasFPU = FALSE`, then your code should use the routines provided by the Standard Apple Numeric Environment (SANE). The routines which SANE provide are covered in the *Apple Numerics Manual.*

So What Can These Chips Do?

The MC68881 and MC68882 are floating–point coprocessors which implement the IEEE standard for binary floating–point arithmetic. The two chips are fully interchangeable and are primarily for use as coprocessors to the MC68020 and MC68030 central processors. The two chips will work as peripheral processors to the MC68000, MC68008, and MC68010 central processors.

Both chips have eight 80–bit general purpose floating–point data registers (`FP0-FP7`), 67–bit arithmetic units with precision greater than the extended format, 67–bit barrel shifter, 46 instructions, trigonometric and transcendental functions, and 21 constants. The MC68882 also has the capability of concurrent execution of multiple floating–point instructions.

Internal Registers for a Higher Capacity to Think

There are eleven separate registers in these puppies: eight data registers, one control register, one status register, and one address register.

Data Registers

There are eight 80–bit floating–point data registers labeled `FP0-FP7`. The extended format, which is used by these registers, will be covered later. When using the FPU from an MPW C and Pascal

application, you can us FP0–FP3 for temporary storage without saving and restoring their values. If you wish to use FP4–FP7 in your assembly routine, then you must save these registers at the start of your assembly code and restore them before you leave the assembly code.

Control Register (FPCR)

Below is a representation of the control register. For the most part, there is no need for you to do anything to the control register directly. It is used internally for determining precision, rounding, and error checking.

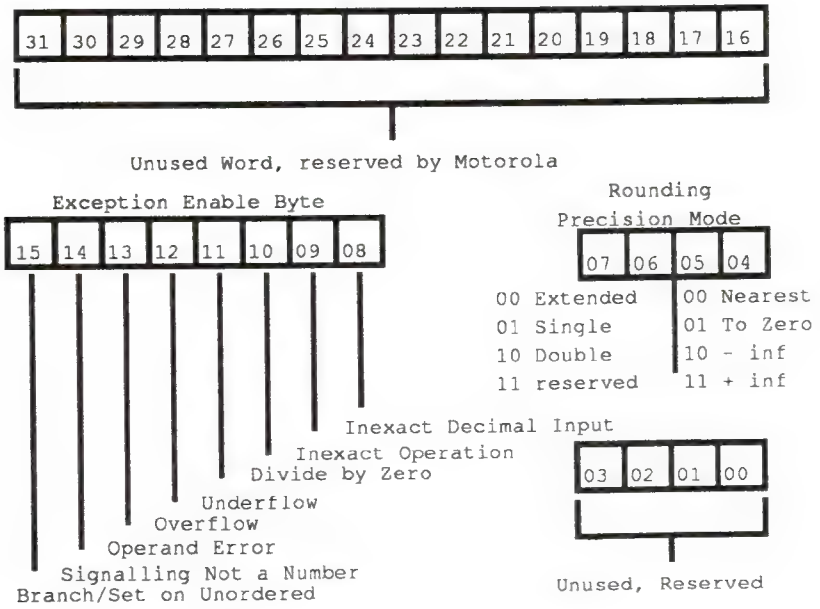


Figure 1–Control Register

Status Register (FPSR)

The status register is diagrammed in Figure 2. This register is also used mostly for internal chores. The condition–code byte is set at the end of each arithmetic instruction. The condition–code byte is translated into a data type; Table 1 shows the relationship between condition codes and data types. The condition code is also used to determine logic equates. If you wish to determine if two numbers are equal, than the Compare statement (FCMP) will check the condition code. Table 2 shows the relationship between the condition codes and logic equates.

The quotient byte is set at the completion of FMOD (Modulo Remainder) and FREM (IEEE Remainder). This byte can be used before a transcendental function to determine the quadrant of a circle in which an operand resides. The FP–exception status byte is used in conjunction with the exception–enable byte of the control register. The FP–accrued exception byte is used to keep a history of the FP exceptions that have occurred since the last set or clear.

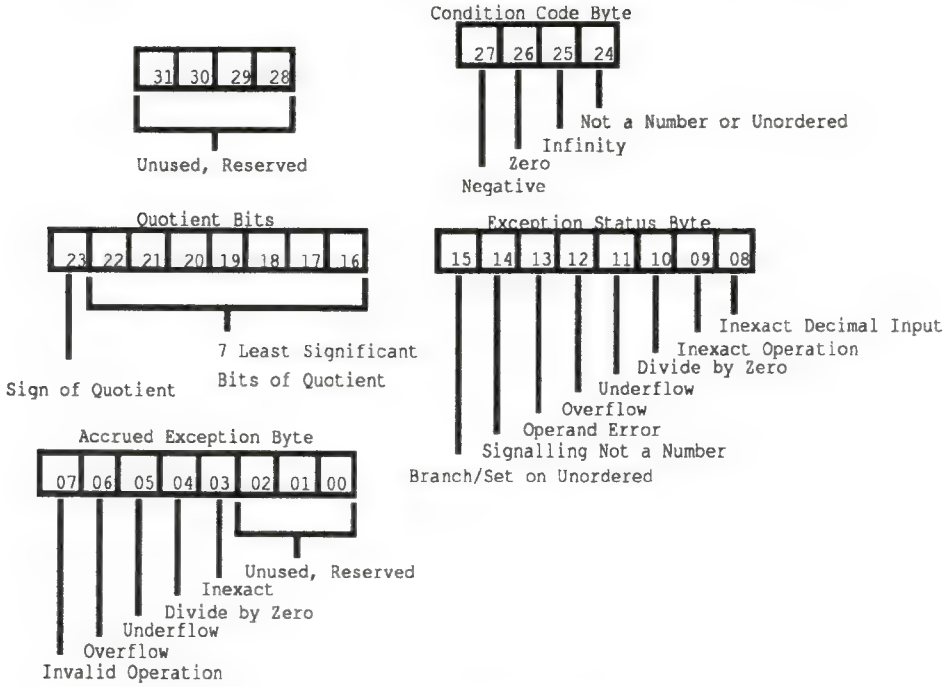


Figure 2–Status Register

Negative	Zero	Infinity	NAN	Result Data Type
0	0	0	0	+ Normalized or Denormalized
1	0	0	0	– Normalized or Denormalized
0	1	0	0	+ zero
1	1	0	0	– zero
0	0	1	0	+ infinity
1	0	1	0	– infinity
0	0	0	1	+ NAN
1	0	0	1	– NAN

Table 1–Condition Code versus Result Data Type

Logic Equate	Abbreviation	Condition Code
Equal to	EQ	Z
Not Equal	NE	not Z
Greater Than	GT or OGT	not(N or NAN or Z)
Not Greater Than	NGT or UGT	NAN or Z or N
Greater Than or Equal	GE or OGE	Z or (not(NAN or N))
Not (Greater Than or Equal)	NGE or UGE	NAN or (N and (not Z))
Less Than	LT or OLT	N and (not(NAN or Z))
Not Less Than	NLT or ULT	NAN or (Z and (not N))
Less Than or Equal	LE or OLE	Z or (N and (not NAN))
Not (Less Than or Equal	NLE or ULE	NAN or (not (N or Z))
Greater or Less Than	GL or OGL	not (NAN or Z)
Not (Greater or Less Than)	NGL or UEQ	NAN or Z
Greater, Less or Equal)	GLE or OR	not NAN
Not (Greater, Less or Equal)	NGLE or UN	NAN
	Oxx is ordered	Z -> Zero
	Uxx is unordered	N -> Negative

Table 2–Logic Equates

Address Register (FPIAR)

Since the coprocessor can do concurrent processing with the MC68020 and MC68030, as well as with itself, the program counter is not necessarily pointing to the logical address of the instruction upon which it is working. So the address register stores the logical address of each floating–point instruction before executing it.

Floating–Point Data Formats

There are four floating–point numeric formats: single–precision binary real format, double–precision binary real format, eXtended–precision binary real format, and Pack decimal real format (a.k.a., BCD). I have given examples of what the FPU will convert your numbers to. The number which I have used for the four examples is Planck's constant (4.136×10^{-15} eV–sec). Other than the size, the first three formats are very similar. The three formats all have the same conversion method and ordering of information.

Single (S) 32 bit

Single precision is represented by 32 bits of information. The high bit (bit 31) is the sign bit (s). The next byte of information (bits 30–23) is the exponent (e), and the last 23 bits (bits 22–0) are the fraction (f). The bits of information are converted into a floating–point number by the following equation:

$$(-1)^s * 2^{(e-127)} * (2^0 + f)$$

The fraction (f) is the strange value. Each bit in the fraction value represents a negative exponent of two. So if bit 22 and bit 16 are high, and all the rest of the bits are low, than the fraction would equal 0.5078125 or ($2^{-1} + 2^{-7}$). So when I give the FPU the number 4.136e–15, it converts the number into the hexadecimal number $04F1503DE, which, in the above equation, looks like:

$$(-1)^{0}*2^{(79-127)}* 2^{0}+2^{-3}+2^{-5}+2^{-7}+2^{-14}+2^{-15}+2^{-16}+2^{-17}+2^{-19}+2^{-20}+2^{-21}+2^{-22}$$

This number is than converted back to a base ten number as 4.13600004803759899e–15. As you can see, the number is correct up to the seventh decimal place.

Double (D) 64 bit

Double precision is represented by 64 bits of information. The high bit (bit 63) is the sign bit (s), The next 11 bits of information (bits 62–52) are the exponent (e), and the last 52 bits (bits 51–0) are the fraction (f). The bits of information are converted into a floating–point number by the following equation:

$$(-1)^{s} * 2^{(e-1023)} * (2^{0} + f)$$

When I give the FPU the number 4.136e–15 as a double, it converts the number into the hexadecimal number $03CF2A07BBC5ED155. This number is than converted back to a base ten number as 4.136000000000000015e–15. As you can see, the number is correct up to the fifteenth decimal place.

EXtended (X) 96/80 bit

Extended precision is represented by 96 bits of information; SANE and FP data register use 80–bit extended numbers, but the FPU extended numbers are 96 bits with 16 unused bits, so the two are basically the same. The high bit (bit 95) is the sign bit (s), The next 15 bits of information (bits 94–81) are the exponent (e), there are 16 unused bits (bits 80–64), and the last 64 bits (bits 63–0) are the fraction (f). The bits of information are converted into a floating–point number by the following equation:

$$(-1)^{s} * 2^{(e-16383)} * (2^{0} + f)$$

When I give the FPU the number 4.136e–15 as a extended, it converts the number into the hexadecimal number $03FCF(0000)9503DDE2F68AA66F. This number is than converted back to a base ten number as 4.136e–15. This number is correct to about the nineteenth decimal place.

Pack Decimal Real (P) BCD Format 96 bits

Pack Decimal Real is represented by 96 bits of information. The bits of these numbers are represented as follows:

bit 95 Sign of Mantissa
bit 94 Sign of Exponent
bit 93–92 used for +–infinity and NANs,otherwise zero
bits 91–81 10–bit Exponent (3 digit exponent)
bits 80–68 unused, zero
bit 67–0, 68 bit Mantissa (17 digit mantissa)

When I give the FPU the number 4.136e–15 as a PDR, it converts the number into the hexadecimal number $401500041360000000000000. This hexadecimal number is filled into the above bit as follows:

bit 95 Sign of Mantissa	0 (binary)
bit 94 Sign of Exponent	1 (binary)
bit 93–92 used for +–infinity and NANs,otherwise zero	00 (binary)
bits 91–80 11–bit Exponent (3 digit exponent)	000000010101 (binary)
bits 79–68 unused, zero	000000000000 (binary)
bit 67–0 68 bit Mantissa (17 digit mantissa)	41360000000000000 (hex)

This number is than converted back to a base ten number as 4.136e–15. This number is correct to the seventeenth decimal place.

So What Tools Do I Have to Play With?

There are four types of opcodes which the math coprocessors support: moves, monodic, dyadic, and miscellaneous conditions. When a coprocessor operation is executed, the first operation which the coprocessor performs is to convert the data to the internal extended precision format, and when the operation is completed, the data is converted to the destination data format.

Moves

The first type which I will describe are the move opcodes. Below is a list of the various formats in which the move commands come.

```
Move
FMOVE.<fmt>     <ea>, FPn
FMOVE.<fmt>     FPm, <ea>
FMOVE.X         FPm, FPn

Move Multiple
FMOVEM <ea>, FP0 - FP3/FP7
FMOVEM FP2/FP4/FP6, <ea>      ;the registers are always moved as 96 bit extended
                              ;data without conversion
Move Register
FMOVE.L         <ea>, FPCR    ;move to control register
FMOVE.L         FPCR, <ea>    ;move from control register

Move Constants from ROM to floating-point register
FMOVECR.X       #ccc, FPn     ;see Table 3 for #ccc

Save and Restore Machine State
FSAVE           <ea>          ;virtual machine state save
FRESTORE        <ea>          ;virtual machine state restore
```

<ea> is a main processing unit (MPU) effective address operand (any 68xxx addressing mode).
<fmt> is the data format size (Byte, Word, Long, Single, Double, eXtended, Packed decimal).
FPm and FPn are floating–point data registers.

#ccc	Mathematical Representation	Numeric Representation
$00	pi	3.14159265358979324
$0B	log(base 10)(2)	0.301029995663981195
$0C	e	2.71828182845904524
$0D	log(base 2)(e)	1.442695040888963410
$0E	log(base 10)(e)	0.434294481903251828
$0F	zero	0
$30	ln(2)	0.693147180559945309
$31	ln(10)	2.302585092994045684
$32	10^0	1
$33	10^1	10
$34	10^2	100
$35	10^4	10,000
$36	10^8	100,000,000
$37	10^16	10,000,000,000,000,000
$38	10^32	100...(28 more zeros)...00
$39	10^64	100...(60 more zeros)...00
$3A	10^128	100...(124 more zeros)...00
$3B	10^256	100...(252 more zeros)...00
$3C	10^512	100...(508 more zeros)...00
$3D	10^1024	100...(1020 more zeros)...00
$3E	10^2048	100...(2044 more zeros)...00
$3F	10^4096	100...(4092 more zeros)...00

Table 3–Constants

Monodic

A monodic operation has one operand. The operand may be a floating–point data register or an MPU effective address. The result is always stored in a floating–point data register. The syntax for monodic operations is listed below:

```
Fxxxx.<fmt>    <ea>, FPn
Fxxxx.X        FPm, FPn
Fxxxx.X        FPn
```

where: <fmt> is (B,W,L,S,D,X,P)
xxxx is one of the Trigonometric (SIN), Transcendental (ATANH), Exponential (ETOXM1), Misc. commands (NEG)

Dyadic

A dyadic operation has two operands. The first operand can be in a floating–point data register, or an MPU effective address. The second operand is the contents of a floating–point data register. The result of the operation is stored in the second operand. The syntax for dyadic operations is listed below:

```
Fxxxx.<fmt>    <ea>, FPn
Fxxxx.X        FPm, FPn
```

where <fmt> is (B,W,L,S,D,X,P)
xxxx is a arithmetic (ADD), compare (CMP)

Condition operations

There are four condition operations: branch (FBcc), decrement and branch (FDBcc), set according to condition (FScc), and trap on condition (FTRAPcc).

Why and How do I Program for a 68882?

Any code which runs on a 68881 will run on a 68882 and vice versa. You do not need to take special care to program for the 68882, but if the chip is available, than special care can noticeably improve the speed of your code. Figure 3 demonstrates the difference between code run on a 68881 and the same code run on a 68882. The 68882 is completely finished running before the 68881 has even started executing the FMOVE instruction. The extra work which you need to do to take advantage of the concurrent processing is fairly minimal.

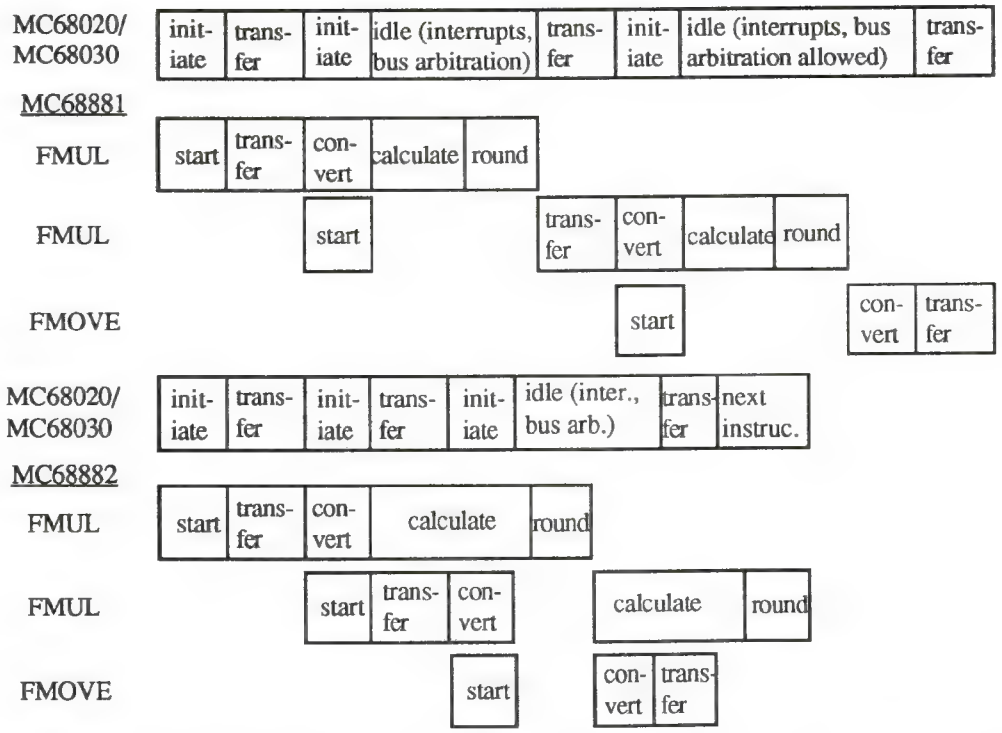


Figure 3–Concurrent Execution versus Non–Concurrent Execution

Before you jump right in and start writing code, you need to understand that there are three different levels of concurrency. The first level is the minimum concurrency operations. These are operations which cannot run concurrently with other operations. Most of these operations are non–floating–point format operations. The minimum concurrency operations are listed in Table 4.

Instruction	Operand Syntax	Operand Format
FMOVE	<ea>, FPn	B,W,L,P
	FPm, <ea>	B,W,L
	FPm, <ea>	P
	FPm, <ea>	P
	<ea>, FPcr	L
	FPcr, <ea>	L
FMOVECR	#ccc, FPn	X
FMOVEM	<ea>, <list>	L,X
	<ea>, Dn	X
	<list>, <ea>	L,X
	Dn, <ea>	X
FTST	FPm	B,W,L,P
F<monodic>	<ea>, FPn	B,W,L,P
F<dyadic>	<ea>, FPn	B,W,L,P
FSINCOS	<ea>, FPc:FPs	B,W,L,P

Table 4–Minimum Concurrency

The next level of operations are the operations which can share some of the FPU time with other operations, these are the partial concurrency operations and they are listed in Table 5. The partial concurrency operations include most of the floating–point format operations.

Instruction	Operand Syntax	Operand Format
FTST	<ea>	S,D,X
	FPm	X
F<monodic>	<ea>, FPn	S,D,X
	FPm, FPn	
F<dyadic>	<ea>, FPn	S,D,X
	FPm, FPn	
FSINCOS	<ea>, FPc:FPs	S,D,X
	FPm, FPc:FPs	X

Table 5–Partial Concurrency

The highest level of concurrency is the fully–concurrent operations which are listed in Table 6. The only operations which can run fully concurrently are the FMOVE operations. There are certain guidelines which you need to follow in order to achieve full concurrency, these guidelines are outlined in Table 6. The most important rule to follow is to avoid register conflict. There are basically two type of register conflict. The first is when the destination register of an operation is the source register of the following operation, and the following operation is a fully–concurrent operation:

```
FADD.<fmt>     <ea>, FP0
FMOVE.<fmt>    FP0, <ea>      ;FP0 conflicts
```

The second type of register conflict occurs when the destination register of an operation is the destination register of the following operation, and the following operation is a fully–concurrent operation:

```
FADD.<fmt>     <ea>, FP0
FMOVE.<fmt>    <ea>, FP0      ;FP0 conflicts
```

where <fmt> is S, D, or X

Instruction	Syntax	Format	No Concurrency	Partial Concurrency
FMOVE	FPm, FPn	X	a	b,c,f
FMOVE	<ea>, FPn	S,D,X		b,c,f
FMOVE	FPm, <ea>	S,D	a	b,d,e
FMOVE	FPm, <ea>	X	a	b

a: Register conflict of FPm with preceding instruction's destination FP data register
b: NAN, unnormalized or denormalized data type
c: Rounding Precision in FPCR set to Single or Double
d: INEX2 bit in FPCR EXC byte is enabled
e: An overflow or underflow occurs
f: Register conflict of FPn with preceding instruction's destination FP data register

Table 6–Fully Concurrent

The next most important optimization rule is to unroll loops. If you properly unroll your loops, than you will be able to eliminate more of the register conflicts. Most loops are designed to do one iteration of a set of instructions. This means that each iteration of the loop is accomplishing one iteration of the set of instructions. If you unroll the loop, then each iteration of the loop can accomplish two or more iterations of the set of instructions. Figures 4 and 5 demonstrate how to unroll your code. The second version (Figure 5) is 25–30 percent faster than the first.

```
                MOVE.L      #count,D0

LOOPTOP         FMOVE.X     <ea_Xi>, FP3
                FNEG.X      FP3
                FETOX.X     FP3
                FMOVE.X     FP3,FP4             ;conflict
                FSUB.X      <ea_Xi>, FP3
                FNEG.X      FP4
                FSUB.X      #1, FP4
                FDIV.X      FP4,FP3
                FNEG.X      FP3
                FADD.X      <ea_Xi>,FP3
                FMOVE.X     FP3, <ea_Xi>        ;conflict
                DBRA        D0, LOOPTOP
```

Figure 4–Newton–Raphson's Method

$X_{i+1} = X_i + f(X_i)/f'(X_i) : f(X) = \exp(-x) - x$

```
                MOVE.L      #count,D0
                FMOVE.D     <ea_Xi>, FP0

LOOPTOP         FNEG        FP0,FP3
                FETOX       FP3
                FMOVE       FP3,FP4             ;conflict
                FSUB        FP0,FP3
                FNEG        FP4
                FSUB.X      #1,FP4
                FDIV        FP4,FP3
                FSUB        FP3,FP0
                DBRA        D0, LOOPTOP

                FMOVE.D     FP0, <ea_Xi>
```

Figure 5–Newton–Raphson's Method (resister–based, unrolled)

$X_{i+1} = X_i + f(X_i)/f'(X_i) : f(X) = \exp(-x) - x$

Conclusion

The last comment which I have to make is for code which is to run during interrupt time. If you plan to use the math coprocessor during interrupt time, you must call `FSAVE` at the start of your routine and `FRESTORE` at the end of your routine. If you do not make these calls and you interrupt another program which is using the FPU, then the other program will not find the FPU in the same state that it was in before the interrupt, and this causes certain death. For more information, refer to Technical Note #235, Cooperating with the Coprocessor.

If you made it this far, and you are still awake, then you should be already to start writing assembly routines for your code which will speed up your math–intensive programs. Just remember that before you try to use the code, you need to check `hasFPU` with a call to `_SysEnvirons`, and if the machine does not have an FPU, then use an alternate SANE version of the math code.

Further Reference:

- *Apple Numerics Manual*, Second Edition
- *Motorola MC68881/MC68882 User's Manual*
- Technical Note #129, _SysEnvirons: System 6.0 and Beyond
- Technical Note #235, Cooperating with the Coprocessor

Macintosh
Technical Notes

Developer Technical Support

#237: TextEdit Record Size Limitations Revisited

Written by: Mary Walsh June 1989

This Technical Note describes another limit on the length of a TextEdit record that was previously undocumented.

The TextEdit chapters in *Inside Macintosh* document the 32K character limit on a TextEdit record length. They do not, however, discuss the more subtle constraint on the size of the `destRect`. By definition, the `destRect` uses integer values for the top-left and bottom-right boundary points. It is possible to have values too large for the `destRect` without reaching the `teLength` limit.

The `nLines` field gives the number of lines in the edit record, and the `lineHeight` field specifies the vertical distance from the ascent line from one line of text to the ascent line of the next line. In styled TextEdit, the `lineHeight` may vary for each line depending on the font and font size. These values are entries in the `lineHeight` table.



Figure 1–LineHeight

The product of the `lineHeight` (or the largest `lineHeight` value in styled TextEdit) and `nLines` gives a good approximation, in pixels, of the vertical dimension of the `destRect` of the `TERec` used. If this value is greater than 32,768, then unpredictable and erratic behavior may result.

For example:

> 2,400 lines of Chicago 12 point yields `nLines` = 2,400 and `lineHeight` = 16.
> `nLines` * `lineHeight` = 2400 * 16 = 38,400. This is above the 32K limit.
>
> 1,200 lines of Times 24 point yields `nLines` = 1,200 and `lineHeight` = 30.
> `nLines` * `lineHeight` = 1200 * 30 = 36,000. This is above the 32K limit.

In both of the examples above, the number of characters in the edit record was less than 32,768.

Check All Constraints

Both `TELength` and the size of the `destRect` must be under the 32K limit. You can compute an approximate vertical height of the `destRect` by finding the product of `nLines` and `lineHeight` (or the largest `lineHeight` value in styled TextEdit).

Further Reference:

- *Inside Macintosh*, Volumes I-371 & V-259, TextEdit
- Technical Note #203, Don't Abuse the Managers

Macintosh
Technical Notes

Developer Technical Support

#238: Getting a Full Pathname

Revised by: Keith Rollin
Written by: Keith Rollin

October 1989
June 1989

This Technical Note describes how to generate a full pathname, given either a Working Directory ID or a real `vRefNum` and a `DirID`. By using the techniques shown in this Note, you can find the full pathname from information such as that returned by Standard File.
Changes since June 89: Added a note on how to check for A/UX. Fixed bug in C version: `BlockMove()` parameters were reversed in `pStrcpy()`; added range checking to `pStrCat()`; changed references from "longint" to "long". Fixed bug in Pascal and C versions: Changed `fsRtDir` to `fsRtdirID` and made references to `gHaveAUX` consistent.

This Note presents two routines. The first routine is called `PathNameFromWD`. It takes an HFS Working Directory ID and returns the full pathname that corresponds to it. It does this by calling `_PBGetWDInfo` to get the `vRefNum` and `DirID` of the real directory. It then calls `PathNameFromDirID` and returns its result.

`PathNameFromDirID` takes a real `vRefNum` and a `DirID` and returns the full pathname that corresponds to it. It does this by calling `_PBGetCatInfo` for the given directory and finding out its name and the `DirID` of its parent. It then performs the same operation on the parent, sticking **its** name onto the beginning of the first directory. This whole process is continued until we have processed everything up to the root directory (identified with a `DirID` of 2).

Warning

This Note is being released in response to demand from developers. However, for the following reasons, generating full pathnames is highly **discouraged**:

- Problems arise when accessing volumes that use file systems other than HFS. For instance, `PathNameFromDirID` uses a butcherous hack to be A/UX friendly. A/UX likes subdirectories separated by slashes in a pathname, rather than colons. This routine automatically uses colons or slashes as separators based on the value of `gHaveAUX`. To check for the presence of A/UX, examine bit 9 of `HWCfgFlags`. If it is set, you are running under A/UX. This global **must** be initialized correctly for this routine to do its thing. However, because of this dependency on the idiosyncrasies of file systems, generating full pathnames for other than display purposes is **discouraged**; it changed in the past when A/UX was implemented, and it may **change** again in the future to support other file systems such as ProDOS, MS-DOS, or OS/2.

- One reason developers have stated for needing to know the full pathname for is so that they can remember the location of a particular file. Saving a file's full pathname should only be used as a **last resort**. Instead, you should remember the

DirID of the directory the file is in along with it's name. This way, you will still be able to find your file even if the directory has been moved. Under System 7.0 or later, save the file's unique 32-bit ID number as well, so that you can also find the file even if it's name has changed.

Either of these methods may fail if a volume has been restored from a backup. In that case, you might be able to find the file by searching with its full pathname. If you find the file, note again the DirID of the directory it is in, and save it for future use. If running under 7.0 or later, also note the file's ID number.

- The routines below are written to return Pascal strings (a length byte followed by the ASCII characters). Hence, the limit on the length of a Pascal string is 255 characters. However, a file's full pathname may be longer than 255 characters. Any routine you write should be prepared for this contingency.

- The reason why the sample routines below were written to return Pascal strings is because that's the way the File Manager likes them. However, as you now know, a file's full pathname may be longer than that acceptable to the File Manager. Therefore, even if you do get fancy and use things like handles and Munger to create a mondo–length filename, you will still have to parse it into pieces less than 255 bytes for the File Manager to handle. Simply using a DirID is a lot easier.

- These routines assume the existence of HFS. If you intend for your program to run under MFS, then you should make the appropriate checks and write special cases accordingly.

MPW Pascal

```
(** PathNameFromDirID ***********************************************************)

     FUNCTION PathNameFromDirID(DirID: longint; vRefNum: integer): str255;

          VAR
               Block: CInfoPBRec;
               directoryName,FullPathName: str255;

          BEGIN
               FullPathName:='';
               WITH Block DO BEGIN
                    ioNamePtr:=@directoryName;
                    ioDrParID:=DirID;
               END;

               REPEAT
                    WITH Block DO BEGIN
                         ioVRefNum:=vRefNum;
                         ioFDirIndex:=-1;
                         ioDrDirID:=Block.ioDrParID;
                    END;
                    err:=PBGetCatInfo(@Block,FALSE);

                    IF gHaveAUX THEN BEGIN
                         IF directoryName[1]<>'/' THEN BEGIN
                              { If this isn't root (i.e. "/"), append a slash ('/') }
                              directoryName:=concat(directoryName,'/');
                         END;
                    END
```

```
                    ELSE BEGIN
                         directoryName:=concat(directoryName,':');
                    END;
                    FullPathName:=concat(directoryName,FullPathName);
               UNTIL (Block.ioDrDirID=fsRtDirID);

               PathNameFromDirID:=FullPathName;
          END;

(** PathNameFromWD ************************************************************)

     FUNCTION PathNameFromWD(vRefnum: longint): str255;

          VAR
               myBlock: WDPBRec;

          BEGIN

               { PBGetWDInfo has a bug under A/UX 1.1.  If vRefNum is a real vRefNum
               { and not a wdRefNum, then it returns garbage.  Since A/UX has only 1
               { volume (in the Macintosh sense) and only 1 root directory, this can
               { occur only when a file has been selected in the root directory (/).
               { So we look for this and hard code the DirID and vRefNum. }

               IF (gHaveAUX) AND (vRefnum=-1) THEN BEGIN

                    PathNameFromWD:=PathNameFromDirID(2,-1);

               END
               ELSE BEGIN

                    WITH myBlock DO BEGIN
                         ioNamePtr:=NIL;
                         ioVRefNum:=vRefnum;
                         ioWDIndex:=0;
                         ioWDProcID:=0;
                    END;

                    { Change the Working Directory number in vRefnum into a real vRefnum }
                    { and DirID. The real vRefnum is returned in ioVRefnum, and the real }
                    { DirID is returned in ioWDDirID. }

                    err:=PBGetWDInfo(@myBlock,FALSE);

                    WITH myBlock DO PathNameFromWD:=PathNameFromDirID(ioWDDirID,ioWDVRefnum)
               END;
          END;
```

MPW C

```
/** PathNameFromDirID ********************************************************/

char *PathNameFromDirID(DirID, vRefNum, s)
     long       DirID;
     short      vRefNum;
     char       *s;
{
     CInfoPBRec      block;
     Str255          directoryName;

     *s = 0;
     block.dirInfo.ioNamePtr = directoryName;
     block.dirInfo.ioDrParID = DirID;
```

```
    do {
        block.dirInfo.ioVRefNum = vRefNum;
        block.dirInfo.ioFDirIndex = -1;
        block.dirInfo.ioDrDirID = block.dirInfo.ioDrParID;

        err = PBGetCatInfo(&block,false);
        if (gHaveAUX) {
            if (directoryName[1] != '/')
                /* If this isn't root (i.e. '/'), append a slash ('/') */
                pStrcat(directoryName,"\p/");
        } else
            /* Append a Macintosh style colon (':') */
            pStrcat(directoryName,"\p:");
        pStrcat(directoryName,s);
        pStrcpy(s,directoryName);
    } while (block.dirInfo.ioDrDirID != fsRtDirID);

    return(s);
}


/** PathNameFromWD ***************************************************************/

char *PathNameFromWD(vRefNum,s)
    long  vRefNum;
    char  *s;
{

    WDPBRec    myBlock;

    /*
    /* PBGetWDInfo has a bug under A/UX 1.1.  If vRefNum is a real vRefNum
    /* and not a wdRefNum, then it returns garbage.  Since A/UX has only 1
    /* volume (in the Macintosh sense) and only 1 root directory, this can
    /* occur only when a file has been selected in the root directory (/).
    /* So we look for this and hard code the DirID and vRefNum. */

    if (gHaveAUX && (vRefNum == -1))
        return(PathNameFromDirID(2,-1,s));

    myBlock.ioNamePtr = nil;
    myBlock.ioVRefNum = vRefNum;
    myBlock.ioWDIndex = 0;
    myBlock.ioWDProcID = 0;

    /* Change the Working Directory number in vRefnum into a real vRefnum */
    /* and DirID. The real vRefnum is returned in ioVRefnum, and the real */
    /* DirID is returned in ioWDDirID. */

    PBGetWDInfo(&myBlock,false);

    return(PathNameFromDirID(myBlock.ioWDDirID,myBlock.ioWDVRefNum,s));
};
```

```
/** pStrcat / pStrCpy ************************************************************/
/*
/*    A couple of utility routines. C is thoughtless enough to not really
/*    support P-strings. In order to perform string copies and concatenations,
/*    these routines are provided.
/*
/********************************************************************************/

#define MIN(a,b)  (((a)<(b))?(a):(b))

char *pStrcat(dest, src)
unsigned char *dest, *src;
{
        long sLen = MIN(*src, 255 - *dest);
        BlockMove(src + 1, dest + *dest + 1, sLen);
        *dest += sLen;
        return (dest);
}

char *pStrcpy(dest, src)
unsigned char *dest, *src;
{
        BlockMove(src, dest, (long) *src + 1);
        return (dest);
}
```

Further Reference:

- *Inside Macintosh*, Volume IV-89, File Manager

Macintosh
Technical Notes

Developer Technical Support

#239: Inside Object Pascal

Written by: Keith Rollin June 1989

This Technical Note briefly explains why Object Pascal and MacApp should only be used to write applications and MPW tools.

Although Pascal can be used to write desk accessories, drivers, XCMDs and other types of stand–alone code, and Object Pascal is an extension of Pascal, Object Pascal cannot be used to write anything other than an application. This limitation is due to the fact that Object Pascal method dispatching relies on a valid `A5` pointing to a jump table. Because MacApp is written in Object Pascal, this limitation applies to it as well.

Once Over Lightly

Object methods cannot always be called directly. To explain why this is so, let's take a case from MacApp. Part of the way MacApp works includes defining `TView` objects that can draw themselves. Whenever an update event occurs, MacApp traverses the list of `TView` objects that are installed in a window and calls the `Draw` method for each one. However, how does Pascal know which `Draw` method to call? Does it call `TYourView.Draw`? Does it call `TView.Draw`? There is no way to know, at compile time, what `TView` objects and descendants of `TView` will be passed to the MacApp update routine. Therefore, there is no way to determine the appropriate `Draw` routine at compile time and generate a direct call to it.

Object Pascal solves this problem by maintaining data structures called Class Info Tables for each Object Class defined. These Class Info Tables not only contain information about the correct procedure to call whenever a message is sent to an object, but they also contain information used to create a new instance of that object.

The mechanism for this dispatching is quite complex and not described here. However, the main point is that the mechanism absolutely relies on special jump table entries. These jump table entries are used to dynamically map method calls to the correct procedure, using the information found in the Class Info Tables. Since desk accessories, drivers, and XCMDs, by their very nature, cannot have a jump table, you cannot use Object Pascal to create them.

Object Pascal **can** be used to write MPW tools, and, in fact, was used to create the MABuild and PostRez tools that come with MacApp 2.0.

Conclusion

For more information on how Object Pascal works, I highly recommend the article by Ken Doyle, "Introduction to Object Pascal," anthologized in *The Complete MacTutor*, Volume 2. However, keep in mind that this information is already slightly out of date, and should not be counted on to be completely accurate at this time. In general, however, it is a good description of what is actually happening when a method call is made.

Further Reference:

- *Inside Macintosh*, Volume II-53, The Segment Loader
- *The Complete MacTutor*, Volume 2, "Introduction to Object Pascal", p. 336
- Macintosh Technical Note #105, MPW Object Pascal Without MacApp
- Macintosh Technical Note #110, MPW: Writing Stand-Alone Code
- Macintosh Technical Note #220, Segment Loader Limitations

Macintosh
Technical Notes



Developer Technical Support

#240: Using MPW for Non-Macintosh 68000 Systems

Written by: Keith Rollin June 1989

This Technical Note discusses using MPW 3.0 for creating software intended to run on 68000-based systems that do not implement the Macintosh run-time architecture. These systems include NuBus™ cards, peripheral devices, and proprietary 68000 systems.

Introduction

Occasionally there is a need to create routines or programs for non-Macintosh systems. Such situations can occur if you are writing a driver for a NuBus board, developing a peripheral that uses a 68xxx microprocessor, or perhaps targeting a proprietary 68xxx machine (Apple uses MPW for all of its ROM and NuBus development.)

For tasks such as this, MPW 3.0 can provide the solution. This Note discusses the problems and issues that arise when doing using MPW 3.0 for this type of development, and it gives some hints and solutions.

To aid you in your efforts, there are several tools available on AppleLink in the Developer Services bulletin board (Developer Techical Support:Macintosh:Tools:Card Dev Tools:) and on *Phil & Dave's Excellent CD*. These tools include utilities to generate checksum data and to prepare your program for downloading.

The following is a brief summary of problem areas you may encounter:

- A5-Relative Globals
- Segmenting and the Jump Table
- ToolBox and OS Routines
- Setting up Your Run-Time Environment

A5-Relative Globals

The Problem

In traditional machine environments, the compiler allocates a certain range of memory in which to store global variables. This memory is established by the machine's memory architecture, and it can usually be referenced by using absolute addressing modes.

Because the Macintosh has a very dynamic run-time environment, programs cannot be written with specific memory locations in mind. Programs are not given a fixed place in memory in which to store their data that will be the same between program invocations. To solve this problem, all Macintosh programs are designed to store global variables in a 32K area pointed to by the 68000

register A5. This could be a problem if your needs require you to reference or store your variables in specific memory locations.

The Solution

This problem can be solved if you are willing to use some macros. A set of macros to do this could look something like the following:

```
#include      <sysEqu.h>

#define pShort              *(short *)
#define pLong               *(long *)
#define lmSFSaveDisk        pShort SFSaveDisk
#define lmCurDirStore       pLong CurDirStore

main()
{
       short foo;
       long bar;

       foo = lmSFSaveDisk;
       lmCurDirStore = bar;

/* or */

       foo = pShort SFSaveDisk;
       pLong CurDirStore = bar;
}
```

Segmenting and the Jump Table

The Problem

When the Macintosh was first developed, memory space was tight. For this reason, a run-time architecture was designed that allowed programs to be divided into segments that could be dynamically loaded and unloaded. Because of this, a program cannot rely on any specific memory locations into which it can be loaded, and hence it has to be freely relocatable. This means that any intra-segment calls (i.e., calls from one routine to another within the same code segment) have to use the PC-relative addressing modes of the 68000. Since these instructions use only signed 16-bit offsets, these branches are limited to a range of 32K bytes. This, in turn, leads us to the historical 32K limit on 'CODE' resource segments. While the restriction in the linker limiting 'CODE' resources to 32K has been lifted with MPW 3.0, it does not resolve the issues with long distance branching.

In order to be larger than 32K, a program should be divided into multiple 'CODE' resource segments. Calls from a procedure in one segment to a procedure in another segment are called inter-segment calls. These calls are performed through a jump table referenced with positive offsets from A5 (Refer to *Inside Macintosh*, Volume II-53, The Segment Loader, for more information on the jump table). The problems that arise from this mechanism are that ROMable code does not get loaded into memory by a Segment Loader, and supporting an A5 jump table may not be desirable.

The Solution

Programs compiled with Pascal or C currently **always** use the 16-bit PC-relative address mode when generating branch instructions. There is no way to change that. However, there are several ways you can get around it:

- Implement your own A5 world
- Use islands for long jumps
- Use assembly language

1) Implement an A5 world in your device that mimics the Macintosh's as closely as possible. This is probably the easiest solution. First, you will be able to program in a normal Macintosh style and not have to take into account considerations that are presented in solutions #2 and #3. It will also allow you to compile and link your program without having to specify any special options.

   After this has been done, and you are ready to download your program to its destination, you can run your program through a filter that: a) determines the final locations of all of the 'CODE' resource segments in the file, and b) creates a jump table with the addresses correctly resolved. In essence, this would be the same as a Macintosh program with all of its segments loaded in memory at the same time.

   Let's take a look at an example. Assume that you have developed a program that is about 40K long, and you would like to have it loaded at location $1000. Because of its length, it is divided into two segments. You have one routine in 'CODE' = 1 that is referenced from 'CODE' = 2 and three routines in 'CODE' = 2 that are referenced from 'CODE' = 1. All of these routines will generate jump table entries. In addition, a jump table entry is generated for the main entry point of your program, as per the Segment Loader chapter of *Inside Macintosh.* This gives us a total of five jump table entries in our program. The file created with MPW would look something like the following:

```
'CODE' = 1

00000000:          main()
...                ...
000038B4:          importantRoutine1()
...                ...
000049F0           End of segment

'CODE' = 2

00000000:          importantRoutine2()
...                ...
00003D0F:          importantRoutine3()
...                ...
00004969:          importantRoutine4()
...                ...
00005892           End of segment
```

'CODE' = 0 (the jump table)

```
00000000:
00000008:  $20 bytes of overhead
00000010:
00000018:
00000020:  00 00 3F 3C 00 01 A9 F0      ; dc.w $0000 / MOVE.W #1,-(A7) / _LoadSeg
00000028:  38 B4 3F 3C 00 01 A9 F0
00000030:  00 00 3F 3C 00 02 A9 F0
00000038:  3D 0F 3F 3C 00 02 A9 F0
00000040:  49 69 3F 3C 00 02 A9 F0
```

When we create our downloadable image, the routines that we are interested in will end up at these locations:

```
main()                  $0000 1000     ($1000 + $0000)
importantRoutine1()     $0000 48B4     ($1000 + $38B4)
importantRoutine2()     $0000 59F0     ($1000 + $49F0 + $0000)
importantRoutine3()     $0000 96FF     ($1000 + $49F0 + $3D0F)
importantRoutine4()     $0000 A359     ($1000 + $49F0 + $4969)
```

Therefore, we should modify our Jump Table to look like this:

'CODE' = 0 (the jump table)

```
00000000:
00000008:  $20 bytes of overhead
00000010:
00000018:
00000020:  00 01 4E 59 00 00 10 00      ; dc.w $0001 / JMP $0000 1000
00000028:  00 01 4E 59 00 00 48 B4
00000030:  00 02 4E 59 00 00 59 F0
00000038:  00 02 4E 59 00 00 96 FF
00000040:  00 02 4E 59 00 00 A3 59
```

2) For some reason, it may be impossible or undesirable to segment your code in Macintosh fashion. You may be importing source code from somewhere else, or you may not be able to utilize a jump table. In cases like this, where your program has to be compiled as one segment, you will hit problems if it is a large program. The Pascal and C compilers will still limit you to branches smaller than 32K. In the cases where you need to execute long distance jumps, the only thing you can do is create "islands" that allow you to make several short hops to your destination. For instance, if it turns out that you are writing a C program which needs to call a procedure that is 70K away, you will have to break up the branch into three smaller ones as follows:

```
main()

[ ... some random code ... ]

procedureNearTheBeginningOfMyProgram()
{
    ...
    Island1(); /*Calling importantButFarAwayRoutine() */
    ...
}

[ ... 20K of intervening code ... ]
```

```
Island1()
{ Island2(); }

[ ... 30K of intervening code ... ]

Island2()
{ importantButFarAwayRoutine(); }

[ ... 20K of intervening code ... ]

importantButFarAwayRoutine()
{
    ...
}
```

3) If programming little islands into your program is too gross for you to comtemplate, then program using the 68xxx assembler, eschewing the high-level compilers. This will allow you to use the absolute addressing mode directly, avoiding the fact that the compilers will not use them. It will also allow you to store into and access fixed memory locations more easily. The following shows some ways of doing this:

```
test    Main
        import test5
        import test6

        org $1000

        jsr test2               ; test of an intraprocedure call just
        jmp test2               ; a few bytes away.

        jsr test3               ; test of an intraprocedure call a
        jmp test3               ; significant number of bytes away.

test2
        jsr (test4).l           ; test of an intraprocedure call more
        jmp (test4).l           ; than 32K away.

; The following instructions won't work on a 68000, but will on a
; 68020 or better. They demonstrate a better alternative to the above
; method, in that they generate PC-relative branching. In order to
; use them, include "MACHINE MC68020" in your assembly source code.

;       bsr.l test4             ; test of an intraprocedure call more
;       bra.l test4             ; than 32K away.

        lea (test4).l,A0        ; alternate test of a > 32K jump
        jmp (A0)

        ds.b 17000              ; padding to force > 16K jump

test3
        jsr test5               ; test of an interprocedure call a
        jmp test5               ; significant number of bytes away

        jsr (test6).l           ; test of an interprocedure call more
        jmp (test6).l           ; than 32K away

        ds.b 17000              ; padding to force > 32K jump

test4
        rts
        endp
```

```
test5   proc
        entry test6

        rts

        ds.b 17000              ; padding to force > 32K jump

test6
        rts

        endp
        end
```

Toolbox and OS Routines

The Problem

Because your program will be operating in a non-Macintosh environment, you will not be able to make any ToolBox or operating system calls. This would not seem to be a problem until you consider that the library routines you are calling my be making such calls themselves. For example, `malloc()` and most `stdio` calls fall into this category.

The Solution

Don't use our libraries. Use your own. Most of the MPW library routines are "clean," but the low-level routines that they rely on use the Toolbox or OS. Identifying those low-level routines that call the Macintosh operating system, determining all the high-level routines that depend upon them, and then programming around them is too difficult a task to undertake. Even if it were done, you would still have to contend with routines that allocated global variables. The best thing to do is avoid our libraries altogether and just write your own.

Setting Up Your Run-Time Environment

The Problem

The Pascal and C compilers do some hidden work to initialize the run-time environment before the part of your application that you have written is actually executed. It is possible that you may wish to take advantage of this setup or may need to duplicate it in order to get your program to execute.

The Solution

With Pascal, most of this initialization is automatically inserted into your main procedure. There is very little you can do about it except to put all of your Pascal routines into separately compiled UNITs and write your entry point in C or assembly.

In the case of C, this initialization is performed by a routine in the file CRuntime.o called `CMain()`. The following is a description of what happens to your source code from the time the C compiler gets it to the time the code you have written is executed:

- MPW C compiles all of the source files and creates object files for the linker. All functions are compiled in **exactly** the same way, including `main()`.
- These files are linked together. If you do not link with the file CRuntime.o, these routines will link together, but they will not have an entry point; the linker will not

have any routine explicitly defined as the first one to be called, and it will default to setting up the first routine that it finds as the entry point.

- If you **do** link with the file CRuntime.o, then you will be linking with a routine called `CMain()`. This routine is marked as being an Entry routine, and it will be the routine that is executed when you launch the Macintosh program.
  - `CMain()` performs the following steps:
    1. Call `_RTInit` (runtime init)
    2. Call `setjmp()`
    3. Check the result of `setjmp()`. If <> 0, go to 6.
    4. Call `main()`
    5. Call `exit()` with result from `main()`.
    6. `RTS`
    - This is what `_RTInit` does:
      1. Call `_DataInit()`.
      2. Save the return address back to whomever ran this program.
      3. Check to see if launched by MPW. If not, then setup `argv` and `argc` to indicate the name of the program with no parameters.
      4. If launched under MPW, initialize some things so that the run-time environment will integrate with MPW. **Calls the Memory Manager**, so make sure that this part of the code is never executed. This is not likely to happen, as `_RTInit` checks and validates several memory locations before it gets this far.
      - This is what `_DataInit()` does:
        1. Assume that `A5` is valid, and that there is data appended to the end of `DataInit` that is used to initialize the globals. This will be done by the linker automatically.
        2. Determine the size of the globals and zero it out.
        3. Read the data at the end of the procedure and use it to initialize the globals. Normally, this process will attempt to use `_BlockMove` on sufficiently large blocks of data, and a small loop for small blocks of data. A version of `DataInit()` that does not call `_BlockMove` is available from Macintosh Developer Technical Support. However, this limits you to 64K of contiguous pre-initialized storage.
    - This is what `exit()` does:
      1. Call any user installed exit procedures.
      2. If called from MPW, set the value of `{Status}`
      3. Determine if `setjmp()` was ever called. If so then call `longjmp()` with a value of 1.
      4. If `setjmp()` was never called, then return directly to the process caller, as saved in step two of `_RTInit`.

While MPW was designed with creating Macintosh programs in mind, it can also be used to write software for non-Macintosh targets. After resolving such issues as creating an appropriate run-time environment, making sure that Toolbox calls are not made, and being aware of the 32K limit for branches and jumps, you should be able to use the high-level Pascal and C compilers. By using assembly language, you should even be able to avoid the problems that they pose.

Further Reference:

- *Inside Macintosh*, Volume II-53, The Segment Loader
- Technical Note #220, Segment Loader Limitations

NuBus is a trademark of Texas Instruments

Macintosh
Technical Notes

Developer Technical Support

#241: Script Manager's Pixel2Char Routine

Revised by: Mark B. Johnson & Dave McGary
Written by: Sue Bartalo

August 1989
June 1989

This Technical Note discusses the `Pixel2Char` routine provided by the Script Manager.
Changes since June 1989: Clarified information, corrected minor errors, and replaced the illustration.

The `leftSide` flag in the `Pixel2Char` routine was inappropriately named, and it should now be called the `leadingEdge` flag. The reason for this change is that the value `Pixel2Char` returns indicates whether a mouse-down occurred on the leading edge of a character, which is not always the left side. (In Arabic or Hebrew, both of which are right-to-left scripts, the mouse-down occurs on the right side of the character.)

With this change, the interfaces also change. Following are both the old and new definitions in Pascal and C respectively:

Old Definition

```
FUNCTION Pixel2Char(textBuf: Ptr;textLen: INTEGER;slop: INTEGER;pixelWidth: INTEGER;
                    VAR leftSide: BOOLEAN): INTEGER;

pascal short Pixel2Char(Ptr textBuf,short textLen,short slop,short pixelWidth,
                        Boolean *leftSide)
```

New Definition

```
FUNCTION Pixel2Char(textBuf: Ptr;textLen: INTEGER;slop: INTEGER;pixelWidth: INTEGER;
                    VAR leadingEdge: BOOLEAN): INTEGER;

pascal short Pixel2Char(Ptr textBuf,short textLen,short slop,short pixelWidth,
                        Boolean *leadingEdge)
```

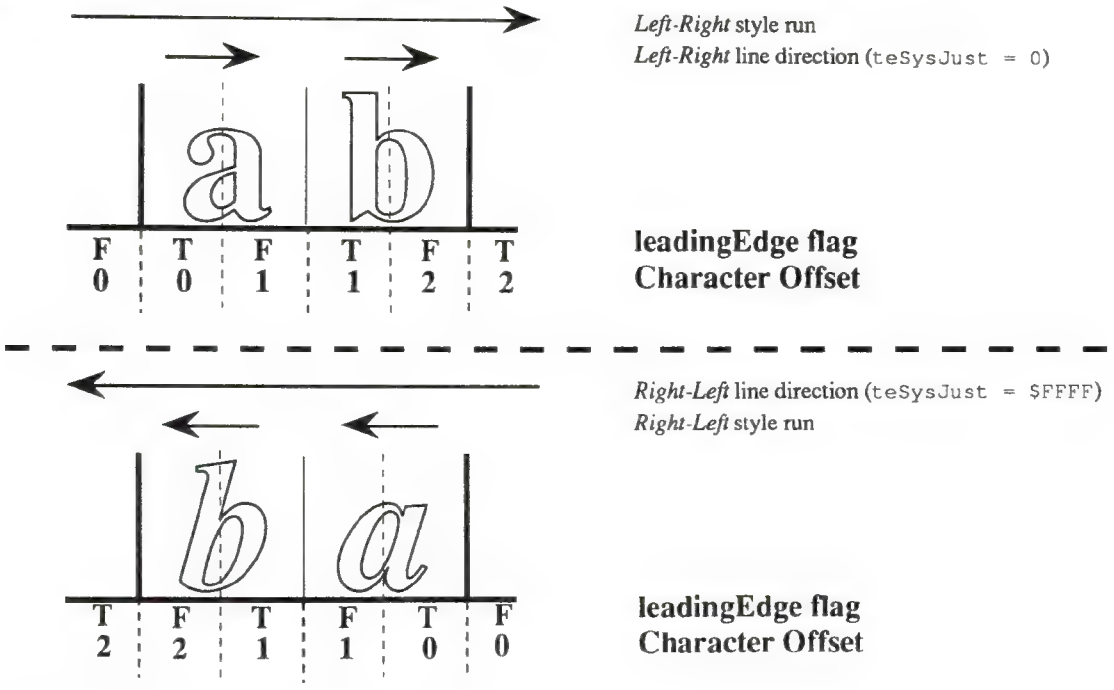
The value of the `leadingEdge` flag is `True` if a mouse-down occurs on the leading edge of the character in its direction (e.g., the left side for a left-to-right script (Roman) and the right side for a right-to-left one (Arabic or Hebrew)). Figure 1 illustrates these differences.

This Note describes the way script systems should work; however, in some systems, the values of `leadingEdge` and character offset are undefined when the `pixelWidth` is outside the boundaries of the text.

You define the start of a right-to-left line to be on the right and the end to be on the left; therefore, it follows that the start of a left-to-right line is on the left, while the end is on the right. The values at the **start** of a line should be `False` for `leadingEdge` and zero for character offset. The values at the **end** are `True` for `leadingEdge` and the character offset is the total byte count of that line.

You should check for these out-of-bounds conditions explicitly and perform the appropriate actions. This change will appear in the next version of the Script Manager documentation.

Character and Line Direction Equal



Character and Line Direction Unequal

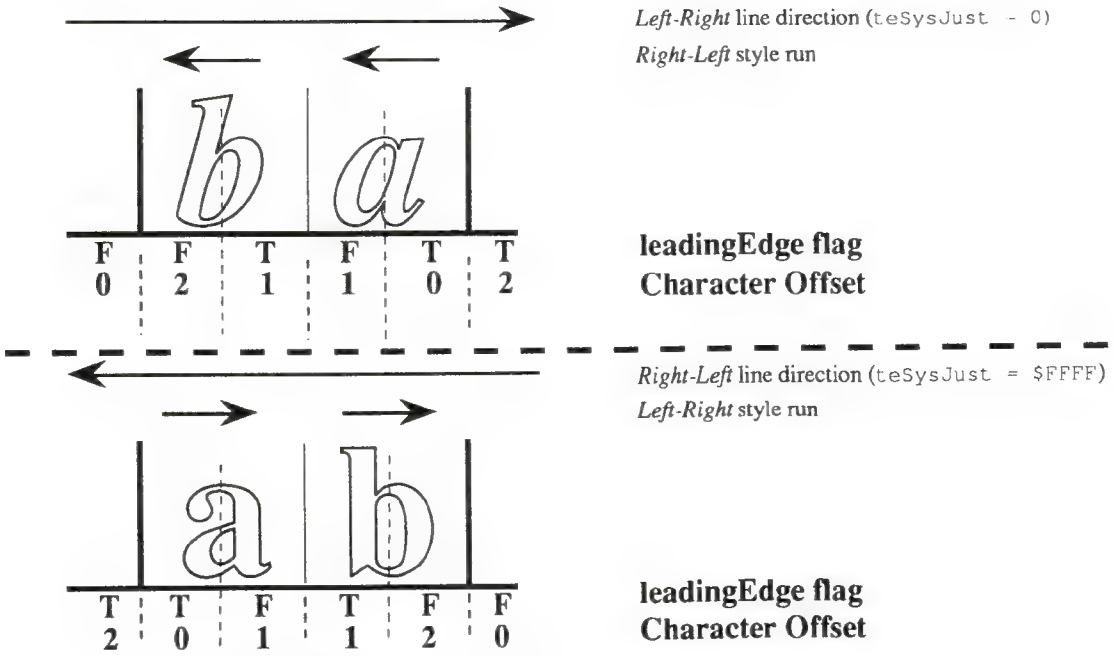


Figure 1–Pixel2Char leadingEdge Flag

Macintosh
Technical Notes

Developer Technical Support

#242: Fonts and the Script Manager

Written by: John Harvey & Peter Edberg June 1989

This Technical Note describes how the Script Manager uses the font family ID to determine a script code.

The traps `_FontScript`, `_IntlScript`, and `_Font2Script` all use a font family ID to determine the script interface system code that they return. This Note describes the process, the way the Script Manager renumbers the Chicago font for non-Roman systems, and the equation for calculating Script IDs from font family IDs.

On a Roman system the Chicago `'FOND'` is numbered zero, but this causes no confusion since Chicago is also the system font. Non-Roman systems must renumber Chicago so that it will not interfere with the mapping of `'FOND' ID = 0` to the correct system `'FOND'`. Typically Chicago is renumbered to 16383.

In *Inside Macintosh*, Volume V-293, The Script Manager, the descriptions of `_FontScript`, `_IntlScript`, and `_Font2Script` state that the current font identification number (e.g., `'FOND'` ID) is used to calculate the correct script code. The equation for calculating script codes from `'FOND'` IDs is as follows:

```
script =((FONDid - $4000) DIV 512) + 1
```

For a specific example, consider the Kyoto font which is one of the fonts included in KanjiTalk. Its `'FOND'` ID is 16385. Plugging that value into the equation above, we get: script = ((16385-16384) DIV 512) +1. Which results in a value of one, the script code for the Kanji script system.

Note that this means that script systems other than Roman can only have 512 separate font families. Furthermore, Roman font families (FOND) must not have an ID greater than 16383, and `'FOND'` ID 16383 is reserved for Chicago on non-Roman systems.

So How Do They Work?

`_FontScript`, `_IntlScript`, and `_Font2Script` begin by setting two Script Manager globals, `Forced` and `Default` to false. Then the two special font family (`'FOND'`) numbers zero and one are mapped to the System and Application font.

Next the 'FOND' ID is tested to see if it is an international font. _FontScript and _IntlScript simply take the value out of the txFont field of the current grafPort. _Font2Script uses the value passed to it. The test is simply:

```
IF FONDid < $4000  {16384}
       script is Roman so return 0
ELSE
       script is international so calculate script id using equation described above
```

Once the script code has been determined, the routine looks at the the Script Manager globals FontForce and IntlForce.

If the currently installed script is Roman and fontForce is true, or if intlForce is true and the routine called was _IntlScript, then the value returned will be the current system script. If the installed script is not Roman; the script code calculated will be returned when the routine called was _IntlScript, intlForce is true, and the script code does not equal the system script.

Once the script code to be returned as been calculated, a final check is made to be sure that the script is installed and enabled. If it is not; Roman is returned, and Forced is set to false and Default is set to true.

What's This Forced Stuff?

Two Script Manager globals, fontForce and intlForce, are flags that support compatibility. Turning fontForce on will cause Roman fonts to be interpreted as belonging to the system script. This provides compatibility for applications that hard-code font numbers.

For example, the Arabic script interface system provides a cdev which lets a user turn fontForce on. When a user does this, any Roman fonts will be mapped to an Arabic font. Note this is only a partially effective measure since the user still does not have complete control over fonts.

It should also be noted that if a user sets fontForce on via the cdev, values returned for fonts with family IDs in the range $0002 to $3FFF (Roman 'FOND' ID range) may vary. This is not a good feature for applications that allow mixed text. To avoid this problem, an application can turn the fontForce flag off before calling _Font2Script or _FontScript. The flag value should be saved before turning it off, and restored later.

The intlForce flag determines how the call IUGetIntl behaves. If this flag is on, IUGetIntl will always return the international resources ('itlx' where x is 0-2) corresponding to the system script. When intlForce is off, the font in the current port will be used to determine which international resources will be returned. This flag lets an application control what date formats, sorting routines, etc. will be used.

For that reason, before calling any of the international utility routines or using the binary to decimal routines, an application should verify that thePort and thePort^.txFont are set correctly, or that intlForce is set properly.

Let's Look at a Picture

The flowchart in Figure 1 illustrates the operation of `_FontScript`, `_IntlScript`, and `_Font2Script`, and how they are affected by the global flags `fontForce` and `intlForce`.

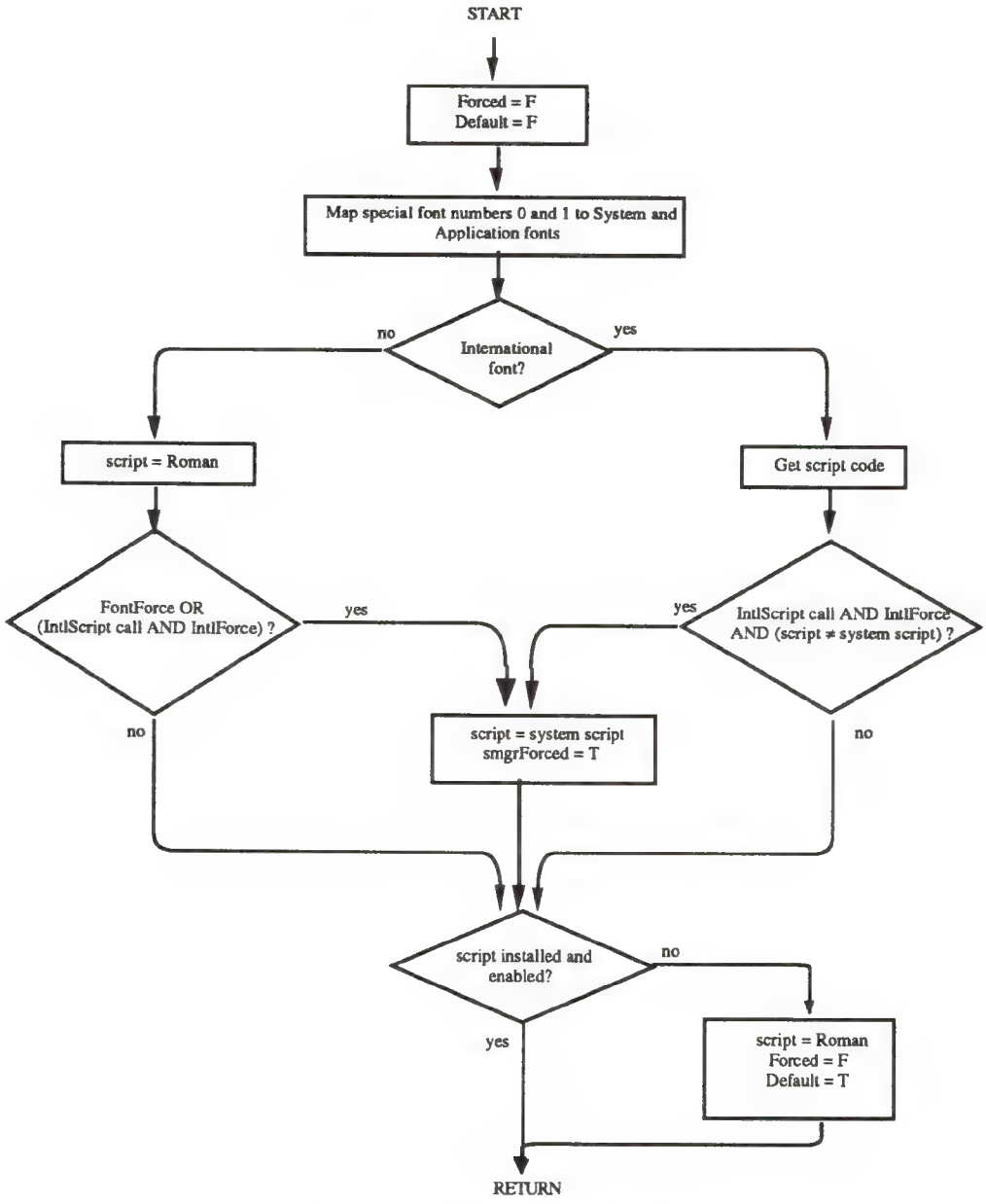


Figure 1–Operation Flowchart

Further Reference:

- *Inside Macintosh*, Volume I-493, The International Utilities Package
- *Inside Macintosh*, Volume V-293, The Script Manager
- *Inside Macintosh*, Volume V-287, The International Utilities Package

Macintosh
Technical Notes



Developer Technical Support

#243: Script Manager Variables

Written by: John Harvey & Peter Edberg June 1989

This Technical Note describes, in detail, the local and global script variables.

Introduction

The Script Manager maintains a number of global variables which can be read with the routine `_GetEnvirons`. These variables can be set by a corresponding routine, `_SetEnvirons`. In addition, each script interface system maintains variables of its own. These are referred to as local variables in Inside Macintosh, Volume V-293, The Script Manager, and are read by `_GetScript` and set by `_SetScript`.

Think of it like this: the Script Manager maintains an environment in which different script interfaces can run. The global variables are used to set up and maintain the environment (thus the names for the routines `_GetEnvirons` and `_SetEnvirons`), and the local variables control how the script itself works (so we have `_GetScript` and `_SetScript`).

Global Variables

When you call `_GetEnvirons` or `_GetScript`, you describe the variable you are interested in with a verb. A verb is simply an integer constant which the Script Manager uses to figure out which variable you want to read or set. The table in *Inside Macintosh*, V-313, gives incorrect names and descriptions for some of the `_GetEnvirons` and `_SetEnvirons` verbs. Table 1 provides correct descriptions.

Constant	Value	Meaning
smVersion	0	Script Manager version number
smMunged	2	Global modification count
smEnabled	4	Script count; 0 if Script Manager not enabled
smBidirect	6	Bidirectional script flag
smFontForce	8	Force font flag
smIntlForce	10	Force international utilities flag
smForced	12	Current script forced to system script
smDefault	14	Current script defaulted to Roman script
smPrint	16	Print action vector
smSysScript	18	Preferred system script
smLastScript	20	Last keyboard script
smKeyScript	22	Keyboard script
smSysRef	24	System folder volRefNum
smKeyCache	26	[Obsolete, do not use]
smKeySwap	28	Keyboard swapping resource handle
smGenFlags	30	General flags
smOverride	32	Script override flags
smCharPortion	34	Ch vs Sp Extra proportion, 4.12 fixed

Table 1–Verbs for _GetEnvirons and _SetEnvirons

The descriptions in the table are still a bit sketchy. The next section describes each variable in more detail and describes the size of each global.

Byte or word globals are mapped to the low-order byte or word of the LongInt returned by _GetEnvirons, with the high-order parts set to zero. Similarly, for these globals _SetEnvirons ignores all but the appropriate part (low-order byte or word) of its params value.

Verb Name	Bytes	Brief Description
smVersion	2	Script Manager version number

At boot time, the version global is initialized to the value SMgrVers. The high byte is the major version number and is defined in the MPW interface files. The low byte is updated when any changes are made to the Script Manager.

smMunged	2	Global modification count

The munged global is initialized to zero at boot time and incremented when:

- _KeyScript changes the key script and updates smKeyScript and smLastScript
- _SetEnvirons is used to change a Script Manager global

smEnabled	1	Script count; 0 if Script Manager not enabled

At boot time or switch-launch time, the enabled global is initialized to zero, then incremented for each script that is installed and enabled. Since the Roman script system should always be installed by the Script Manager, a value of zero indicates that the Script Manager is not enabled.

It should be noted that older versions of the Script Manager treated this as a `Boolean`. In other words, if there was more than one script installed, `_GetEnvirons(smEnabled)` would return 255 (when `_GetEnvirons` returns a `Boolean` value $FF represents true).

For this reason, when testing to see if more than one script is installed, it is best to test as follows:

```
scriptsinstalled := GetEnvirons(smEnabled);
IF scriptsinstalled > 1 THEN
{more than one script available, use Chartype, etc.}
```

`smBidirect`	1	Bidirectional script flag

The bidirectional global indicates that at least one bidirectional script is installed. It should be set to true ($FF) by the Arabic and Hebrew script systems. This is not presently done, but will be corrected in future versions of these systems.

`smFontForce`	1	Force font flag
`smIntlForce`	1	Force international utilities flag
`smForced`	1	Current script forced to system script
`smDefault`	1	Current script defaulted to Roman script

At boot time, `FontForce` and `IntlForce` are set from the `'itlc'` resource, and `Forced` and `Default` are set to zero. These are all flags with the value zero for false and $FF for true. `FontForce` and `IntlForce` control the operation of the `_FontScript`, `_Font2Script`, and `_IntlScript` routines. `Forced` and `Default` report the actions of these routines.

Setting `FontForce` to true forces Roman fonts to be interpreted as belonging to the system script. This is for compatibility with applications that hard-code font numbers.

`IntlForce` determines the behavior of the `_IUGetIntl` call. When `intlforce` is set to true, `_IUGetIntl` will return a handle to the international resources (of type `'itlx'` where `x` is 0-2) for the system script. When `IntlForce` is false, the `_IUGetIntl` will use the font of the current port to determine the appropriate resources to fetch. Thus date formats, sorting, etc. can reflect the current script.

`smPrint`	4	Print action vector

Print action routine vector; set up at boot time. See Technical Note #174, Accessing the Script Manager Print Action Routine.

`smSysScript`	2	Preferred system script
`smLastScript`	2	Last keyboard script
`smKeyScript`	2	Keyboard script

At boot time and switch-launch time, `SysScript` and `KeyScript` are set from the `SysScript` field of the `'itlc'` resource if that script is installed and enabled; otherwise, `SysScript` and `KeyScript` are set to Roman (without setting `Default`).

The KeyScript global is the current keyboard script, tested and updated by the _KeyScript routine. When _KeyScript changes KeyScript, it moves the old value to LastScript. _KeyScript can also swap the current key script with the last one, which it retrieves from LastScript. The KeyScript value is also used to get the proper keyboard script icon and to retrieve the proper 'KCHR'.

SysScript specifies the system script, and is used, for example, by _FontScript, _Font2Script, and _IntlScript.

KeyScript, LastScript, and SysScript always contain integers that correspond to a script number. Script numbers are documented in The Script Manager chapter of *Inside Macintosh*, Volume V-293.

smSysRef	2	System folder volRefNum

Set from the global BootDrive at boot time and switch-launch time. SysRef was originally a way of testing for vanilla launch versus switch launch; now the Enabled global is used for that purpose.

smKeyCache	2	[Obsolete, do not use]
smKeySwap	4	Keyboard swapping resource handle

The 'KSWP' resource handle is put here at boot time and switch-launch time. A 'KSWP' resource contains a table of key sequences that will cause the currently installed 'KCHR' (keyboard mapping table) to change to the preferred system 'KCHR', switch to the Roman 'KCHR', or rotate among the available 'KCHR' resources. The table includes the virtual key code and the modifier keys. The following is the 'KSWP' resource for the Kanji script interface system.

```
resource 'KSWP' (0, sysheap) {
       {/* array: 3 elements */
              /* [1] */
              Rotate, 49, controlOff, optionOff, shiftOff, commandOn,
              /* [2] */
              System, 70, controlOff, optionOff, shiftOff, commandOn,
              /* [3] */
              Roman, 66, controlOff, optionOff, shiftOff, commandOn
       }
};
```

The resource says rotate 'KCHR' resources if a Space–Command key occurs, switch to the system 'KCHR' on keypad plus (+)–Command key, and switch to the Roman 'KCHR' on keypad asterisk (*)–Command key.

smGenFlags	4	General flags

Only the two high-order bits are defined (in the file ScriptEqu.a), as follows:

smfShowIcon = 31	(show icon even if only one script)
smfDualCaret = 30	(use dual caret for mixed direction text)

The high-order byte of smgrGenFlags, containing these flags, should be setup from the flags byte in the 'itlc' resource. This is not presently done, but will be fixed in future versions of the Script Manager.

The following MPW Pascal procedure demonstrates how to get script 'SICN' resources to display even if there is only one script system installed.

```
PROCEDURE SetSICN;
VAR
        SICNstate: Longint;
        err: Oserr;

BEGIN
        SICNstate := GetEnvirons(smGenFlags);
        BSET(SICNstate,smfShowIcon);

        err := SetEnvirons(smGenFlags,SICNstate);
END;
```

smOverride	4	Script override flags

At present, this is not set or used by the Script Manager. It is, however, reserved for future improvements.

smCharPortion	2	Ch vs Sp Extra proportion, 4.12 fixed

This is 16-bit fixed-point value in 4.12 format (e.g., 10% = $0199). It is initialized to 10 percent at boot time. It is intended to be used by script systems to allocate space among intercharacter spacing and interword spacing when justifying text.

A 16-bit fixed-point value in 4.12 format is similar to the fixed-point number type defined on page I-79 of *Inside Macintosh* . The obvious difference being that it is only 16 bits long. The integer part of the value is stored in the high four bits, and the fractional part is stored in the low 12 bits.

sign bit	4	2	1	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{16}$	$\frac{1}{32}$	$\frac{1}{64}$	$\frac{1}{128}$	$\frac{1}{256}$	$\frac{1}{512}$	$\frac{1}{1024}$	$\frac{1}{2048}$	$\frac{1}{4096}$

16-bit Fixed-Point Number in 4.12 format

Local Variables

Every script interface system has local variables. Page V-132 of *Inside Macintosh* lists verbs which are constants that indicate which variable you want to read or set. The table of constants used to access the local variable, although more accurate than the global table, does contain a few inaccuracies. In addition four new constants have been added. Table 2 gives the correct constants.

Constant	Value	Meaning
smScriptVersion	0	Script Interface version number
smScriptMunged	2	Local modification count
smScriptEnabled	4	Script Enabled Flag
smScriptRight	6	Right to Left Flag
smScriptJust	8	Justification Flag
smScriptRedraw	10	Word Redraw Flag
smScriptSysFond	12	Preferred System Font
smScriptAppFond	14	Preferred Application Font

smScriptNumber	16	Script 'itl0' ID
smScriptDate	18	Script 'itl1' ID
smScriptSort	20	Script 'itl2' ID
smScriptFlags	22	Script Flags Word (**new**)
smScriptToken	24	'itl4' ID number (**new**)
smScriptRsvd	26	Reserved
smScriptLang	28	Script's language code (**new**)
smScriptNumDate	30	Number/date representation codes (**new**)
smScriptKeys	32	Script 'KCHR' ID
smScriptIcon	34	Script 'SICN' ID
smScriptPrint	36	Script printer action routine
smScriptTrap	38	Trap entry pointer
smScriptCreator	40	Script file creator
smScriptFile	42	Script file name
smScriptName	44	Script name

Table 2–Local Variable Constants

Here again the descriptions are a little terse. The following section describes each variable in more detail and describes the size of each variable.

Verb Name	Bytes	Brief Description
smScriptVersion	4	Script Interface version number

When the script interface is loaded, this is set to the current version number.

smScriptMunged	2	Local modification count

This variable is incremented each time _SetScript is called.

smScriptEnabled	1	Script Enabled Flag

A Boolean which indicates whether the script has been enabled. Set to $FF when enabled and zero when not enabled.

smScriptRight	1	Right to Left Flag

A Boolean indicating if text should be drawn right to left or left to right. It is set to $FF for right to left text (Arabic and Hebrew scripts) and zero for left to right (Roman).

smScriptJust	1	Justification Flag

A byte flag which describes how text should be justified. The possible settings correspond to the justification flags used by TextEdit.

0 = left justification
1 = center justified
-1 = right justified

`smScriptRedraw` 1 Word Redraw Flag

A byte flag describing how much of a line should be redrawn when text is being entered.

0 Only draw a character
1 Redraw the entire word
-1 Redraw the entire line (Arabic)

`smScriptSysFond` 2 Preferred System Font

This is the font family ID for the preferred System Font. In a Roman system, `ScriptSysFond` is 0, the family ID for Chicago.

`smScriptAppFond` 2 Preferred Application Font

Font family ID for the preferred Application Font. In a Roman system, `ScriptAppFond` is 3, the family ID for Geneva.

`smScriptNumber` 4 Script `'itl0'` ID

Resource ID of `'itl0'` for this script. The `'itl0'` resource describes how numbers and times should be displayed. The resource ID should match the country version code for a given country.

`smScriptDate` 4 Script `'itl1'` ID

Resource ID of the `'itl1'` for this script. The `'itl1'` describes how dates should be displayed.

`smScriptSort` 4 Script `'itl2'` ID

Resource ID of the `'itl2'` for this script. The `'itl2'` contains routines for sorting. See Technical Note #178.

`smScriptFlags` 2 Script flags

This verb provides access to the script flags word, which contains bit flags that describe features of the script. This word is initialized from the script's `'itlb'` resource. Constants specifying the bit numbers are described in Table 3.

Constant	Bit Number	Description
`smsfIntellCP`	0	script has intelligent cut and paste
`smsfSingByte`	1	script has only single bytes
`smsfNatCase`	2	native characters have upper and lower case
`smsfContext`	3	contextual script (e.g., AIS-based)
`smsfNoForceFont`	4	will not force characters
`smsfB0Digits`	5	has alternative digits in B0-B9
`smsfForms`	13	uses contextual forms for letters
`smsfLigatures`	14	uses contextual ligatures
`smsfReverse`	15	reverses native text, right-left

Table 3–Constant Bit Numbers

`smScriptToken`	2	Script `'itl2'` ID

Resource ID of the `'itl4'` for this script. The `'itl4'` contains contains tables needed by the number formatting and conversion routines and the `_intlTokenize` routine. See Script Manager 2.0, Interim Chapter.

`smScriptRsvd`	4	Reserved
`smScriptLang`	2	Script's language code

This verb accesses a word which contains the current language code for the script. The language codes are defined in the MPW interface files.

`smScriptNumDate`	2	Number and date representation codes

This verb accesses a word containing the number and date representation codes for the script. The number representation code is in the high byte of the word, and the date code is in the low byte.

The possible values for number representations and date codes are declared as constants in the MPW interface files.

The number codes are:	and the date codes are:
intWestern = 0;	calGregorian = 0;
intArabic = 1;	calArabicCivil = 1;
intRoman = 2;	calArabicLunar = 2;
intJapanese = 3;	calJapanese = 3;
intEuropean = 4;	calJewish = 4;
	calCoptic = 5;

`smScriptKeys`	4	Script `'KCHR'` ID

Resource ID of preferred `'KCHR'` resource. The `'KCHR'` resource is used to map virtual key codes into the correct character code. See Technical Note #160.

`smScriptIcon`	4	Script `'SICN'` ID

Resource ID of the small icon that is used to represent which country specific resources (`'itl0'`, `'itl1'`, `'itl2'`, `'KCHR'`) are currently installed in the system. Presently, the Roman system does not display the `'SICN'`. Arabic, Kanji, Chinese, and Hebrew interface systems do display this icon in the upper-right corner of the menu bar.

`smScriptPrint`	4	Script printer action routine

Print action routine vector; setup when script is installed by Script Manager. See Technical Note #174, Accessing the Script Manager Print Action Routine.

`smScriptTrap`	4	Trap entry pointer

Pointer to Script dispatch routine. Script Manager routines always belong to one of two groups. The first group of routines are common to every script interface system, and the second group must be supplied by the script interface system. This variable will point to a dispatch routine for the interface-supplied routines. When you call `_ScriptUtil`, it looks at the selector that is

passed and either calls a common routine or calls the routine whose address is stored in `ScriptTrap`. The routine in `smScriptTrap` will then use the selector to vector to the correct routine. In general, routines that display or measure text in some way will be supplied by the interface.

A list at the end of this Note indicates which routines are implemented by the Script Manager and which routines are supplied by a script interface system.

`smScriptCreator` 4 Script file creator

The four character creator type for the script interface's file. For Roman it is "ZSYS," the same creator as any system file has.

`smScriptFile` 4 Script file name

A pointer to the a Pascal string which contains the name of the file containing the script interface system. For the Roman SIS, it is System.

`smScriptName` 44 Script name

A pointer to a Pascal string which contains the script interface's name. For Roman it is naturally, "Roman."

Who Does What?

Table 4 breaks the documented routines into common Script Manager routines and interface specific routines.

Common Routines	Interface Supplied
_FontScript	_CharByte
_IntlScript	_CharType
_KeyScript	_Pixel2Char
_GetEnvirons	_Char2Pixel
_SetEnvirons	_Transliterate
_Font2Script	_FindWord
_Format2Str	_HiliteText
_FormatStr2X	_DrawJust
_FormaX2Str	_MeasureJust
_GetFormatOrder	_ParseTable
_InitDateCache	_VisibleLength
_IntlTokenize	_FindScriptRun
_LongDate2Secs	_PortionText
_LongSecs2Date	
_Str2Format	
_String2Date	
_String2Time	
_StyledLineBreak	
_ToggleDate	
_ValidDate	

Table 4–Script Manager Routines and Interface Specific Routines

`_GetScript` and `_SetScript`, which return the values of local script variables, are implemented by the Script Manager for some verbs and the script interface system for others.

There is also a group of Script Manager routines which don't use the `_ScriptUtil` trap, but are documented in The Script Manager chapter of *Inside Macintosh*, Volume V-293 or The Script Manager 2.0 Interim Chapter. There routines are utilities that read and write to low-memory or PRAM. It is important to use these routines when they are available. That will allow Apple to modify where global variables, etc. are stored, and your application will remain compatible. The utilities are:

```
GetDefFontSize
GetSysFont
GetAppFont
GetMBarHeight
GetSysJust
SetSysJust
ReadLocation (documented in Interim Chapter)
WriteLocation (documented in Interim Chapter)
```

Further Reference:

- *Inside Macintosh*, Volume V-293, The Script Manager
- Technical Note #160, Key Mapping
- Technical Note #174, Accessing the Script Manger Print Action Routine

Macintosh
Technical Notes



Developer Technical Support

#244: A Leading Cause of Color Cursor Cursing

Revised by: Alan Mimms
Written by: Alan Mimms

October 1989
June 1989

Working with color cursors you create from scratch can cause headaches. This Technical Note may help a bit.

Changes since June 1989: Added a warning about purgeable `'clut'` resources.

If you're building an application that creates color cursors, you may encounter some quirks present in Color QuickDraw that manifest themselves in hard-to-understand ways.

If your cursor is, say, 15 pixels tall and 9 pixels wide, you might be tempted to use these values for the `bounds.bottom` and `bounds.right`, respectively, in your cursor's pixel map. **Don't**. The problem is that when the cursor's image needs to be expanded (i.e., when you specify a two bit-per-pixel cursor and the mouse pointer is on an eight-bit screen) the `_SetCCursor` trap rounds the width of the pixel map in such a way that you'll get only the space required for a 15 by 8 pixel map allocated for the expanded cursor data. When the cursor's image is expanded into this too-small expanded cursor data handle as a 15 by 9 pixel map, something in your heap will get munched.

The cure is simple. Make **certain** that you always specify that the `pixmapHandle^^.bounds` be 16 by 16. This will cause `_SetCCursor` to properly allocate the expanded data area, and all will be well in the land. Since the amount of data **drawn** for a cursor is specified by the cursor's pixel values and `'clut'` resource, trying to save a few bytes by making the bounds rectangle smaller than 16 by 16 wouldn't have been very helpful anyway.

Another potential problem is with the color cursor's color table. If you load the color table from a `'clut'` resource using `_GetCTable`, you should make sure that the `'clut'` is marked non-purgeable while the color cursor is in use. If you do not take this precaution, bombs will occur if your `'clut'` gets purged at in inopportune time.

Macintosh
Technical Notes

Developer Technical Support

#245: Font Family Numbers

Revised by: Bill Mitchell
Written by: John Harvey

March 1991
August 1989

This Technical Note discusses the range of numbers available for identifying font families, how they are allocated among script systems, and what numbers should be used for fonts that were designed to be used as a tool in an application.

Changes since August 1990: Apple Computer no longer registers font family ID numbers; this note has been altered to reflect the change in policy. Also, the relationship of outline fonts to font families is discussed briefly.

Introduction

The `txFont` field in a `GrafPort` record is a signed word. This means that there are 32,768 positive numbers that can be used to identify font families (0 through 32,767). Currently these numbers are broken into groups and assigned to different script interface systems. Apple has also created a pseudo-script called `smUninterp`. This pseudo-script provides a range of numbers that can be used to identify fonts that are used as tools in an application. An example of this is the font used by MacPaint® for its palette symbols.

A Brief History of Fonts Resources and Font Family IDs

There are, as of System 7.0, four types of resource for storing font data and font family information: `'FONT'`, `'NFNT'`, `'FOND'`, and `'sfnt'`.

When the Macintosh was first introduced, all font data was stored in the `'FONT'` resource. Fonts with the same typeface but different sizes were grouped together into families by storing a unique family ID in bits 7-14 of the font's resource ID and the font's point size in bits 0-6. The family was named by including a `'FONT'` resource with a point size of zero. This method is documented in *Inside Macintosh*, I-234, The Font Manager. A disadvantage of this system was that, since the font family ID had to fit into eight bits, the range of numbers available was only 0-255; 0-127 were reserved for Apple, and 128-255 were available for third-party developers.

With the arrival of the 128K ROMs came two new resources, `'NFNT'` and `'FOND'`, which are documented in Inside Macintosh Volume IV, chapter 5, The Font Manager. The `'NFNT'` has the same internal format as the old `'FONT'` resource, but may use any resource ID, since it is associated to its family not by its resource ID, but rather by a table in a new resource, the `'FOND'`. The `'FOND'`, which stores size-independent family information, contains a family name and a font association table (*Inside Macintosh*, IV-39) with entries for each `'FONT'` or `'NFNT'` resource in its family. Each font association table entry includes a word to hold a specific font's point size, a word for its style, and a word for its associated `'FONT'` or `'NFNT'` resource ID.

Since font families were now identified by a 'FOND' ID, not by eight bits in a 'FONT' ID, the range of font family IDs was expanded to 0 through 32,767.

The addition of outline font support saw the introduction of the 'sfnt' resource, whose internal format differs substantially from the older font data resources ('FONT' and 'NFNT') and is documented in *The TrueType Font Format Specification*, APDA catalog number M0825LL/A. As with the 'FONT' and 'NFNT' resource types, the 'sfnt' is grouped into a font family by an entry in the font association table of the 'FOND'; such an entry always has a listed point size of zero.

Scripts and Their Ranges

As stated in the introduction, each script interface system that can run on the Macintosh has a range of font family IDs assigned to it. A script's range can be calculated (see Macintosh Technical Note #242, "Fonts and the Script Manager"), but a table of scripts, their script IDs, and the range of font family IDs assigned to each script are provided in this Note.

Script	Script ID	Font Family IDs
System Reserved	All	0 - 1*
Roman	0	2 - 16382
System Reserved	0	16383
Japanese	1	16384 - 16895
Chinese	2	16896 - 17407
Korean	3	17408 - 17919
Arabic	4	17920 - 18431
Hebrew	5	18432 - 18943
Greek	6	18944 - 19455
Russian	7	19456 - 19967
Reserved	8	19968 - 20479
Devanagari	9	20480 - 20991
Gurmukhi	10	20992 - 21503
Gujarati	11	21504 - 22015
Oriya	12	22016 - 22527
Bengali	13	22528 - 23039
Tamil	14	23040 - 23551
Telugu	15	23552 - 24063
Kannada	16	24064 - 24575
Malayalam	17	24576 - 25087
Sinhalese	18	25088 - 25599
Burmese	19	25600 - 26111
Cambodian	20	26112 - 26623
Thai	21	26624 - 27135
Laotian	22	27136 - 27647
Georgian	23	27648 - 28159
Armenian	24	28160 - 28671
Maldivian	25	28672 - 29183
Tibetan	26	29184 - 29695
Mongolian	27	29696 - 30207
Ethiopian	28	30208 - 30719
Non - Cyrillic Slavic	29	30720 - 31231
Vietnamese	30	31232 - 31743
Sindhi	31	31744 - 32255
Uninterpreted Symbols	32	32256 - 32767

* Font family IDs 0 to 1 are reserved. The system always maps the correct system font to font 0 and the correct application font to font 1. On a Roman system, Chicago is the system font and Geneva is the application font. Obviously this is not true on the Kanji system or any other non-Roman script interface system (see Technical Note #242, Fonts and the Script Manager for more details on how various script systems map fonts to font family 0).

Breakdown of the Roman Script Range

The 16,384 IDs available for Roman systems are further delimited as follows:

0 to 255	Font families which are named in the method described in *Inside Macintosh*, Volume 1, The Font Manager. These IDs should no longer be used. Please note that Apple's system fonts (e.g., Chicago, Geneva, New York, etc.) will always retain their old IDs. Also note that IDs 0 and 1 are especially inviolate, as the system has the right to map any font family from any script system to these IDs.
256 to 1023	Reserved numbers. These numbers should be thought of as reserved space that the Font/DA Mover can use to resolve past and future font ID conflicts. Numbers in this range should not be used as a font family's original ID.
1024 to 16382	Commercial fonts. A particular Roman font family should always fall somewhere in this range, though its exact position may vary, or be altered during the copying of fonts, to avoid conflict with other families.
16383	**Reserved.** This number should not be used.

On non-Roman script systems, the system font in that script will be ID 0. On such systems, Chicago is mapped to 16383 rather than 0; the system software will always look at 16383 rather than 0 for Chicago in these cases.

Developers who use a font as a method of storing symbols which are used in a palette, or store a font in the resource fork of their application for some other special purpose, should use numbers in the range 32,256-32,767. This range is not associated with any script.

The History of Font Registration

Obviously, there are a lot more font family IDs available for Roman systems than for any other. This situation is fortunate, because there are a lot more fonts designed and sold for Roman languages than for any other. In fact, today there are many more fonts than unique font family IDs, and so many families may be using the same ID, causing potential conflicts among them.

In an attempt to alleviate the `'FOND'` ID conflict problem, Macintosh Developer Technical Support implemented a font registration program in early 1989. The registration program provided ID numbers only to those font families which were entirely composed of `'NFNT'` resources. Families which consisted of `'FONT'` resources **and** `'NFNT'` resources were not registered.

As of late 1990, all free font family IDs had been registered, so any new font families created after that time must share their ID with a registered family; Developer Technical Support no longer registers family IDs.

Ask For It by Name

The paucity of unique 'FOND' IDs is not as catastrophic as it sounds. Since April 2, 1988 (See Macintosh Technical Note #191, "Font Names"), Apple Computer has recommended that developers make all references to font families by name, not number. With that document's description of how to store font information by name in applications and documents, the only remaining major hurdle to 'FOND' ID independence was that the 'PICT' resource still identified fonts by family number; with System 6.0.5, however, a new 'PICT' opcode was added to allow 'PICT's to reference font data by 'FOND' name.

As of this writing, nearly all major applications identify families by name, and since Font/DA Mover and System 7.0 renumber conflicting families when copying fonts, there should never be confusion except when using newer fonts on old software which references by number.

Notes on the Location of Font Resources

Certain restrictions apply when storing font resources in an application's resource fork. Due to the order of preference with which the font manager looks for font resources, you should make sure your application font does not use the same name as any system font resource. Otherwise, the system may unexpectedly or intermittently choose the system's version of the font over yours, particularly under System 7.

Fonts should **never** be stored in a document's resource fork. If you close a font-laden document, the system will retain references to memory which was deallocated when the document closed; the system could then alter memory at those locations and corrupt the heap. In addition, System 7 does not support the storing of fonts in document resources. Note that HyperCard stacks **are** documents. If you feel that your stack loses all its artistic merit without a certain font, you should license it for distribution in a suitcase file and let the users install it in their systems.

Fonts are **proprietary**; to include unlicensed fonts with your application violates copyright laws. Contact a font developer for permission before releasing its fonts with your software.

Further Reference:

- *Inside Macintosh*, Volume I, The Font Manager
- *Inside Macintosh*, Volume IV, The Font Manager
- *Inside Macintosh*, Volume VI, The Font Manager
- Technical Note #191, Font Names
- Technical Note #198, Font/DA Mover, Styled Fonts, and 'NFNT's
- Technical Note #242, Fonts and the Script Manager
- The TrueType Font Format Specification, APDA catalog number M0825LL/A.

MacPaint is a registered trademark of Claris Corporation.

Macintosh
Technical Notes

Developer Technical Support

#246: Mixing HFS and C File I/O

Written by: Jeanette Robertson August 1989

This Technical Note discusses the problem of mixing calls to the Macintosh file system with calls to MPW C library file I/O routines.

Problems with Communication Between HFS and C

Frequently, developers want to use both Macintosh file I/O and C file I/O. Developers who do this must keep in mind that they are combining two distinct file representations (the Macintosh and ANSI C). The only limitation on mixing HFS and C I/O functions is that they cannot be mixed on the same open file. There are three reasons why this cannot be done.

First, there is no routine that maps between a C `FILE` struct (returned by `fopen()`) to an HFS `fRefNum` (needed to call HFS functions). Similarly, there is no call to create a `FILE` struct given an `fRefNum` returned by `FSOpen()`. Thus, there is no way that the information from an `fopen()` call could be used to do a `fsread()`.

Second, even if the first problem were solved, the C libraries eventually call the HFS file system, but keep some internal state information. So, if you call HFS directly (say, `SetFPos()`), the C file system has no way of knowing a call was made and, therefore, doesn't update its state information.

Similarly, there is no mechanism for synchronizing the C library's buffers. For example, you perform an `fwrite()` with some number of characters which get put into a buffer without flushing it. Then you perform an `FSWrite()` with something else. Neither the C library nor HFS are aware that the other has written to the file.

Simply put, you cannot make HFS calls on a file opened with `fopen()` or `fdopen()`; you cannot use C library I/O on a file opened under HFS. However, here are some points to consider when manipulating the same file using both C and HFS. Keep in mind this isn't frequently done; there may be problems of which we are unaware.

One obvious problem is keeping track of the working directory. Be sure to save and restore the current working directory when moving between HFS and C I/O calls.

Following is an example routine, which mixes HFS and C I/O. Notice that it doesn't really solve the problem of mixing the two file systems, but rather it shows how to use `fopen()` with standard file (working directories or directory IDs) in general.

```
HardRockCocoJoe()
{
        Point                   where;
        char                    *prompt = "\pWe Are Here";
        char                    *fname = "\pHardRockCocoJoe";
        FileFilterProcPtr       fileFilter = NULL;
        short                   numTypes = 1;
        SFTypeList              typeList;
        DlgHookProcPtr          dlgHook = NULL;
        SFReply                 reply;
        OSErr                   result;
        FILE                    *TheFile;
        short                   fileNum;
        long                    numofChars = 10;
        short                   currentVRefNum;

        (void) GetVol(NULL, &currentVRefNum);

        result = FSOpen(fname, currentVRefNum, &fileNum);
        if (result != 0) {
                /* error checking */
        }
        else {
                result = FSWrite(fileNum, &numofChars, "from MacIO");
                if (result != 0) {
                        /* error checking */
                }
        }

        (void) FSClose(fileNum);

        where.h = 80;
        where.v = 90;
        typeList[0] = 'TEXT';

        SFGetFile (where, prompt, fileFilter, numTypes, typeList, dlgHook, &reply);

        result = SetVol(reply.fName, reply.vRefNum /* from sfgetfile */);
        if (result != 0){
                /* error check */
        }

        p2cstr(reply.fName);
        TheFile = fopen (reply.fName, "a+");
        fprintf (TheFile, "\nfromC\n");
        fclose (TheFile);

        result = SetVol(NULL, currentVRefNum);
        if (result != 0){
                /* error check */
        }

        result = FSOpen(fname, currentVRefNum, &fileNum);
        if (result != 0){
                /* error check */
        }
        else {
                numofChars = 12;
                SetFPos(fileNum, fsFromLEOF, 0);
                result = FSWrite (fileNum, &numofChars, "from MacIO 2");
                if (result != 0) {
                        /* error check */
                }
        }
}
```

Assuming the user chooses HardRockCocoJoe from the Standard File dialog box, the result of this routine is a file called HardRockCocoJoe, which contains the following data:

```
from MacIO
fromC
from MacIO 2
```

By keeping track of the working directory, you can work with HFS file I/O and C I/O. Of course, if you are working with many files, it could be a problem keeping track of the correct `paramBlock` and expensive to open and close the files each time you switch.

Another approach would be to construct a pathname from the Macintosh file system that could be passed to the C I/O functions. Technical Note #238, Getting a Full Pathname, goes into complete detail as to how this is done using either a working directory or `vRefNum` and `DirID`. But, this solution has serious drawbacks and is not recommended. One problem is that you have to manually create the pathname as a string and stuff the needed folder separators into that string. The current separator, the colon (:), may change in the future. A bigger problem is the length of the pathname. Currently, it can only be 256 characters, and that may be hard for you to guarantee. Lastly, there could be a problem if the user should change the directory or rename a file.

You Were Warned

Be aware that you are responsible for any file problems you may have mixing HFS and C file I/O. If it can be avoided, by all means, avoid it.

Further Reference:

- *Inside Macintosh*, Volume I, The Standard File Package
- *Inside Macintosh*, Volume IV, The File Manager
- Technical Note #238, Getting a Full Pathname

Macintosh
Technical Notes

Developer Technical Support

#247: Giving the (Desk)Hook to INITs

Revised by: Pete Helme
Written by: Pete Helme

October 1989
August 1989

This Technical Note discusses INIT evils, the foremost of which deals with clearing DeskHook and `DragHook` at INIT time.
Changes since August 1989: Added warning about clearing `DragHook`.

If you've survived the typical DTS Tirade* and still feel the need to display a dialog box or window in an INIT, you need to be aware of a problem which exists on Macintoshes earlier than the Macintosh II (remember those?). There is a low-memory global named `DeskHook` ($A6C), which can contain a pointer to a routine responsible for painting the Macintosh desktop. If it is `NIL`, which is usually the case, the System paints the desktop with the standard pattern.

When you start displaying dialog boxes or windows that obscure the desktop in your INIT (this is really hard **not** to do, so keep reading and don't let us catch you skipping ahead to another Technical Note with pictures of human genetic experiments gone sour—you know which one I'm talking about), the System looks at `DeskHook` for a desktop updating routine. Since the Macintosh II, the System has cleared this hook prior to calling your INIT; however, on machines before the Macintosh II, this hook is not cleared before the System calls your INIT, so there is usually some junk hex lounging about in there. Since `DeskHook` is not `NIL`, when the System tries to use and perform a `JSR` to this "address," it blows big chunks.

So unless you like big chunks, the easy way to fix this problem is to clear `DeskHook` before doing any window drawing. The most logical time to do this is during your initialization:

```
PEA     thePort(A6)          ; initialize own grafPort off A6
_InitGraf
_InitFonts
_InitWindows
_TEInit
CLR.L   -(A7)
_InitDialogs
CLR.L   $A6C                 ; DeskHook
```

For you high-level types, this translates into:

```
procedure ClearDeskHook;
      inline
       $42B8, $0A6C;              { CLR.L $A6C ; DeskHook }
```

It doesn't hurt to clear it on newer machines either, even if it is already clear (you'll just have to trust me on this one), so go ahead and clear it all the time.

Note: Some INITs might actually use `DeskHook`. However, the popular ones that paint a picture on the desktop, which you might think use it, do not. They use other methods. (We know, we checked. We have the technology.) For those of you who have seen a real procedure pointer in location $A6C on earlier Macintoshes, don't worry. The system does not actually set `DeskHook` for its own use until the first application loads, so clearing it while INITs load is okay.

If there is some daring INIT out there which sets `DeskHook`, we haven't heard about it. As is the case with many low-memory globals, using `DeskHook` has never been supported.

Watch Out For This Guy Too

It should also be noted that `DragHook` ($9F6) is not cleared during INIT time on early Macintoshes either, and it will probably contain $FFFFFFFF. I guess no one in early Macintosh System Software wanted `DeskHook` to be lonely. `DragHook` can contain a pointer to a procedure that is called continuously while the mouse button is down. If you have a user interface at INIT time that ultimately calls `_TrackGoAway` for windows or `_TrackControl` for controls, look out. If the control is of the type to allow one of it's parts to be dragged with an outline, like a scroll bar's thumb, it calls `_DragTheRgn`, which checks to see if `DragHook` is `NIL`, and if it is not, it tries to perform a `JSR` to whatever address is there. `_TrackGoAway` also tries to perform a JSR to that address if it's not `NIL`. So make sure `DragHook` is clear before you attempt to use one of these routines.

In fact, if you've got a lot of spare time, like you're on the Voyager 7 project waiting to come into contact with Black Lectroids and you have an old Macintosh 512KE or Plus lying around, why not try randomly clearing out **all** low-memory globals and see what does and doesn't crash? Sure to be an ice breaker at parties.

* (Asterisk)

What am I yakking about? If you've ever written to DTS about getting help with displaying some kind of modal monster at INIT time (remember this for the later quiz kids, that's the time before this sort of thing should normally happen), you know of what I yak.

We have this pet peeve with INITs that interrupt the boot process with a modal dialog box which asks us to enter our name then proceeds to ask us how many characters we just entered and what we had for dinner, especially when we've left the desk to go get a fix of a highly caffeinated substance. INITs were created for developers (and us) to install system patches and device drivers and make the occasional rude startup sound to annoy the person occupying the cube next to ours. INITs were **not** developed to ask for personal (asexual) histories.

We do not mean to say that we don't like all graphics at boot time. The ShowINIT icon mechanism that was popularized by Paul Mercer is great. In fact, we encourage it's use and we gladly give out the ShowINIT MPW object file, with installation help, to anyone who asks for it. This is an excellent method for a developer to inform a user whether a ROM patch or device driver has been successfully installed (show the red X through the icon on the rare occasion when things go wrong). Of course, this doesn't work if some non-social INIT makes an `_InitWindows` call, which wipes clean the entire screen (and with it all previous ShowINIT icons). You may argue that having the `_InitWindows` trap wipe out the entire screen at INIT time is a bad Macintosh OS INIT-time design, but this is one of our biggest complaints with the whole INIT look-at-my-fancy-splash-screen-or-complete-my-insanely-great-modal-dialog-box phenomenon.

If you feel the need to notify the user of an important occurrence during boot time, initialize a notification request with the Notification Manager in your INIT code (see Technical Note #184, Notification Manager, and yes, it is perfectly legal to use at INIT time), and the system will notify the user after the boot process, when the event mechanism starts. The now alerted user can then activate your desk accessory, application, or whatever and you can perform whatever kind of pyrotechnics you want.

If you are going, "But, but, but..." because various Apple products are guilty of INIT evils, then you should realize that we are giving Apple engineers the same, if not more, grief to cleanse their acts as well.

It's not that we're telling you that you cannot put up a modal dialog box at INIT time if you feel like it's really-absolutely-positively-it-was-your-dying-mother's last-wish-necessary. It's just that DTS would like to see a cleaner Macintosh interface (as I'm sure you all would), and a more uniform boot time appearance can help achieve this goal.

Macintosh
Technical Notes

Developer Technical Support

#248: DAs & Drivers in Need of (a Good) Time

Revised by: Pete Helme October 1989
Written by: Pete Helme August 1989

This Technical Note describes a few complications which rear their rather ugly little heads when a desk accessory or driver needs periodic time. It also presents a few solutions to work around these problems and make life easier, at least periodically.

Changes since August 1989: Corrected `_BitClr` and `_BitSet` examples. Okay, I admit it. I was having too good of a time when I wrote the original Note and messed up the bit manipulations at the end. My vision was blurred; I was in no condition to see those tiny little things.

See Jane's Heap, See accRun...

MultiFinder is our friend. Our friend, that is, until a driver or desk accessory is called when in an unknown heap. Then things get complicated. When a driver is called at `accRun` time under MultiFinder, one can never be exactly sure of the heap in which it will find itself. When a DA receives an open call, or any other messages besides accRun, under MultiFinder, the system heap is switched in as the current heap[1]. During the `accRun` cycle, whatever heap is currently switched in will be the driver's heap as well, and surprise, surprise, that heap may not be the system heap.

This situation could be a real problem if your DA allocates memory or creates a window during that `accRun` period. Why? What if the application whose heap the DA is in suddenly slips a bit and decides to call it quits before the DA? You'd be stuck with allocated blocks in a zone that suddenly doesn't exist. Eventually, your DA would go belly up, and whoever bought your DA or driver would be on the phone to a local dealer demanding retribution.

So what's the solution? The easiest way out of this situation is to simply not do any memory allocation or display any newly created windows or dialog boxes during `accRun`. So what if it's a cop out, it's easy to implement.

Being the good souls that we are in DTS, we're not going to leave you hanging there with nowhere to go. We prefer you heed the previous solution, but we realize that there may be rare times when you might need a window during `accRun`. We've devised a solution, albeit a bit strange, but one that's easy enough to use.

[1]This is true unless a user "force" switched the DA into an application heap by holding down the Option key when opening the DA. In this particular case, the application's heap will be switched in.

The basic problem is that the DA needs to know in which heap it should be allocating it's new storage. It would be nice if the DA knew in which heap it was opened and could allocate the new stuff there, and it's easy enough to do, so here is what you need to know to do it.

Switching from the current heap to the "preferred" heap is fairly simple. When you feel the need to allocate memory or create a window during `accRun`, first save the current heap zone with `_GetZone`. Now, get the handle to the actual driver for the DA. You can do this by looking at the `dCtlDriver` offset of the DAs device control entry (DCE). The DCE is always in register `A1` when a control call to the DA is made. Use `_HandleZone` on the handle to the DAs driver to give you a pointer to the heap in which the driver resides. Pass that value to `_SetZone`. Once you have switched in the correct heap, do whatever memory allocation or window creation you need, and then make sure to set the current zone back to the saved zone with `_SetZone`.

The following short routine, borrowed, in part, from an MPW sample DA, shows one way to set up the correct zone.

```
pascal short DRVRControl(CntrlParam *ctlPB, DCtlPtr dCtl)
{
        extern void    doCtlEvent();
        extern void    doPeriodic();

        THz     driverZone;
        THz     savedZone;

        /*
         * The current grafPort is saved & restored by the Desk Manager
         */
        switch (ctlPB->csCode) {
                case ACCEVENT:                          /* accEvent */
                        HLock(dCtl->dCtlStorage);       /* Lock handle since it will
                                                           be dereferenced */
                        doCtlEvent( *((EventRecord **) &ctlPB->csParam[0]),
                                        (Globals *)(*dCtl->dCtlStorage));
                        HUnlock(dCtl->dCtlStorage);
                        break;

                /*
                 * Hey!  Look here!
                 */
                case ACCRUN:                            /* periodicEvent */
                        savedZone = GetZone();          /* save a pointer to the current heap */
                        driverZone = HandleZone(dCtl->dCtlDriver); /* get the heap our driver
                                                                      resides in */
                        SetZone(driverZone);            /* use that as the current heap */
                        doPeriodic(dCtl);               /* go do your periodic stuff */
                        SetZone(savedZone);             /* restore the old heap */
                        break;

                default:
                        break;
        }
        return 0;
}
```

One note of caution: Watch out for changes in the resource chain when in `accRun`, as it may not be what you expect when MultiFinder is active.

"Houston, We've Got a Re-Entry Problem"

Displaying an alert or other modal dialog box is a common occurrence in Macintosh programming, even in DAs. But since DAs are not applications, modal dialog boxes pose other problems when displayed under MultiFinder. This problem is reentrancy. If your DA or driver asks for periodic time, it continues to receive it when it display a modal dialog box. Bummer. Your modal dialog routine might even be called again, and again, and again, and again, and you get the idea. This problem occurs because `_ModalDialog` calls the `_SystemTask` trap, which in turn calls drivers which asked for time, including yours. There is no internal check by the System for this possible problem, so it's up to you and your driver to be prepared.

We realize that some DAs and drivers expect, and depend upon, this functionality. We're just taking this opportunity to inform the rest of you that this is a situation about which you should be aware.

An easy way to avoid this issue is to simply tell the Device Manager not to call your DA when you display an alert or other modal dialog box. Remember that `dNeedTime` bit you set when you opened your DA so you'd get time? Just clear it before your call to `_Alert` or `_ModalDialog`. As long as the bit is clear, your DA does not receive any periodic time. Remember to reset it once you are done with your `_Alert` or `_ModalDialog` trap call.

The `_BitClr` and `_BitSet` Toolbox utilities are a mite on the brain-damaged side, and the bits are the reverse of conventional 680x0 numbering (numbering starts from the high-order bit instead of the low-order bit). This difference necessitates a calculation for figuring out the correct bit as shown in the following example: (I think whoever wrote these Toolbox utilities did this just to see if anyone was paying attention.)

Pascal

```
BitClr(@dce^.dCtlFlags, 2);          { clear bit 5/dNeedTime bit. IM I-471 }
BitSet(@dce^.dCtlFlags, 2);          { set bit 5/dNeedTime bit.  IM I-471 }
```

or the kind of more efficient, but less efficient than C:

```
CONST
dNeedTime = $2000;                   { Bit 5 of high-order byte of word }

dce^.dCtlFlags := BAND(dce^.dCtlFlags, BNOT(dNeedTime));  { clear bit 5/dNeedTime bit. }
dce^.dCtlFlags := BOR(dce^.dCtlFlags, dNeedTime);         { set bit 5/dNeedTime bit. }
```

C

```
BitClr(&dce->dCtlFlags, 2);          /* clear bit 5/dNeedTime bit. IM I-471 */
BitSet(&dce->dCtlFlags, 2);          /* set bit 5/dNeedTime bit.  IM I-471 */
```

or the somewhat more efficient:

```
#define dNeedTime 0x2000             /* Bit 5 of high-order byte of word */

dce->dCtlFlags &= ~dNeedTime;        /* clear bit 5/dNeedTime bit. */
dce->dCtlFlags |= dNeedTime;         /* set bit 5/dNeedTime bit.  */
```

One More Thing...

We cannot overemphasize our viewpoint that if you are writing a DA and the result looks and acts more like an application, then **write an application** instead and save us all a lot of headaches.

Further Reference:

- *Inside Macintosh*, Volume II, The Memory Manager
- Technical Note #180, MultiFinder Miscellanea

Macintosh
Technical Notes

Developer Technical Support

#249: Opening the Serial Driver

Revised by: Sriram Subramanian December 1989
Written by: Sriram Subramanian August 1989

This Technical Note describes the recommended, safe, and compatible way to open the Macintosh serial driver, and it explains why you should no longer check for port availability.
Changes since October 1989: Corrected syntax errors in the sample code.

Starting with the 128K ROM, we recommend that applications do not check the low-memory globals `SPConfig`, `PortAUse`, and `PortBUse` before opening the serial driver. It is no longer the application's responsibility to test for the availability of the serial ports. When running AppleTalk Phase 2, it is now possible to use the printer port for asynchronous serial communication while AppleTalk is active and using an alternate connection, such as EtherTalk or TokenTalk.

The serial driver automatically verifies that the serial port is correctly configured and free for an asynchronous driver; if it is not correctly configured or free, the serial driver returns either the result code `portNotCf` or `portInUse`. The serial driver already has all the code built into it for testing the availability of the serial ports before trying to complete the `_Open` call. Therefore, since all of the required checks are made inside the driver itself, we recommend that a simple `OpenDriver` call be made when you need to use a serial port.

By using just the `OpenDriver` call to the serial driver, you ensure that your code is both user-friendly and compatible with future versions of the System Software.

Pascal

```
result := OpenDriver('.AOut',AoutRefNumber);    { Check result codes in a real application. }
result := OpenDriver('.AIn',AinRefNumber);      { See failure mechanism in Sample Code.     }
```

C

```
result = OpenDriver("\p.AOut",&AoutRefNumber); /* Check result codes in a real application.*/
result = OpenDriver("\p.AIn",&AinRefNumber);   /* See failure mechanism in Sample Code.    */
```

If you must maintain compatibility with the 64K ROMs, call `_SysEnvirons`, then either call `RAMSDOpen` for the 64K ROM machines or `OpenDriver` for the others.

Further Reference:

- *Inside Macintosh*, Volume II-249, The Serial Driver
- *Inside Macintosh*, Volume IV-225, The Serial Driver
- Technical Note #129, _SysEnvirons: System 6.0 and Beyond
- DTS Q & A Stack